Foundations of Clustering: New Models and Algorithms

Yuyan Wang

Committee: Benjamin Moseley (Chair)
Sergei Vassilvitskii
Zachary Lipton
Peter Zhang

Tepper School of Business
Carnegie Mellon University

This dissertation is submitted for the degree of
Doctor of Philosophy in Operations Research

May 2022

Acknowledgements

First and foremost, I would like to express deep gratitude to my advisor, Benjamin Moseley, for leading me into the area of my Ph.D. research. I hardly had a clue about my prospect as a researcher when I first arrived at CMU. I thank him for opening the doors to all possibilities and igniting my passion for research.

I thank Sergei Vassilvitskii, whom I have had the privilege of collaborating with and learning from. I thank him for his insights, support and patient guidance throughout my research life and career seeking, and for helping me become the researcher I am.

I thank Zachary Lipton and Peter Zhang, for serving on my committee and providing insightful feedback, and for helping me rethink about the purpose and structure of my research and the connection it has to real world applications.

I thank my collaborators, Senaka Buthpitiya, Sami Davis, Alex Fabrikant, Silvio Lattanzi, Ravi Kumar, Kirk Pruhs, Alireza Samadian and Rudy Zhou. My collaboration with them has taught me a lot of skills, and has far-reaching meaning apart from the paper publication.

I thank all the faculty in the Operations Research department at the Tepper School of Business. I have benefited from their knowledge and expertise. I thank Lawrence Rapp and Laila Lee for seamlessly running the administration of the PhD program at Tepper.

I thank all the friends who made my Ph.D. experience far more enjoyable. Many thanks to my cohorts: Violet Chen, Sagnik Das, Özgün Elçi, Melda Korkut, Kyra Gan, Su Jia and Thomas Lavastida. Special thanks to Violet and her furry friend, Model, for three years of roommate-ship and all the happiness and tears; and Sagnik Das, for selflessly helping me out through the challenging times of my Ph.D. I also thank Musa Çeldir, Serim Hwang, Anthony Karahalios, Neda Mirzaeian, Yanhan (Savannah) Tang, Ziye Tang and Luxi Wang for all the memories. I thank Yimeng Liu and Yichi Zhang, my friends from China, for their empathy and support, for sharing and connecting to my pains when starting a new chapter in my life.

Lastly, I owe almost everything to my family. I thank my parents, Wenbo Wang and Xiaozhen Yan, for their unconditional love and support; my grandparents, and the other family members. Despite the geographical distance and time difference, they are always there for me. I shall also thank my furry family member, Mochi. How lucky I am to have loved and be loved by a dog like you.

Abstract

In this dissertation, we study clustering, one of the most common unsupervised learning problems. This dissertation covers recent developments in both clustering theory and machine learning practice. In particular, it explores how to bridge the gap between theory and practice by making them benefit from each other.

Many clustering algorithms are developed in an application-oriented way lacking the guidance of theory. For some clustering problems it is hard to mathematically characterize what is being optimized. The arising needs in the ML/AI community, such as fairness and scalability, also require updates in current problem formulations. The first few chapters of this dissertation lay the theoretical foundation for multiple clustering problems. We first establish the formal optimization framework. Such a framework gives us conceptual understanding of the problems and becomes the basis for optimization and algorithm design. We then discuss the performance of existing approaches and come up with new algorithms beating the state-of-the-art. Empirical evaluations also verify that the new algorithms perform better in both quality and efficiency, showing it is beneficial to view these problems through the lens of theory.

We study one classic clustering problem: **hierarchical clustering**. Unlike other more well-formulated clustering problems such as k-means, the theoretical study of hierarchical clustering has kicked off recently. The first chapter focuses on new objective function design for hierarchical clustering on point inputs in a Euclidean space. It provides theoretical guarantees for a popular heuristic. The second chapter studies how to incorporate fairness into the hierarchical clustering optimization framework. It defines fair hierarchical clustering trees and discusses how to design algorithms that find fair solutions for previous hierarchical clustering objectives established by the community. Surprisingly, in this setting fairness could be imposed at minimal loss in clustering solution performance.

The focus is then shifted to speeding up famous clustering algorithms in scenarios where they are known to be inefficient. We consider **average-linkage**. Building the hierarchical tree from bottom to top, it is one of the most commonly used hierarchical clustering algorithm. It is known to scale poorly to large datasets, as it requires iteratively searching for two clusters with the smallest average pairwise distance among the current clustering, which

is time consuming. To speed up the cluster search, we introduce a new technique named "clustering embedding". It maps clusters into points in Euclidean space. The points are then used as surrogates for the clusters, enabling us to apply **Approximate Nearest Neighbors (ANN)** techniques. We reduce the previous quadratic bound on running time to only slightly super-linear.

New challenges could also be imposed by a new data input format other than the conventional sample-feature matrix. Consider **relational database**, one of the most common data storage format that is highly compact. The naive way of running conventional ML algorithms requires converting the given database into the matrix format. This could cause the input size to grow exponentially. Instead, we design algorithms that could work directly on the relational databases without recovering the sample-feature matrix. We give such algorithms for the classical k-means problem. We show how to adapt the famous **k-means++** algorithm and find a constant approximation for the optimal k-means solution.

On the other hand, this dissertation shows how we can rethink the design of combinatorial algorithms by augmenting the algorithms with learned oracles using a data driven approach. Traditional algorithm performance design and analysis is often bottle-necked by worst-case instances. Meanwhile, the practitioners often have historical records about past data and solutions. Training ML oracles on the data can give us knowledge about the current problem instance, and this knowledge could be used to help the algorithm go beyond the "hurdles" of hard instances. We call such knowledge **"predictions"**. The remaining chapters focus on proposing feasible predictions in the contexts of different clustering problems and discussing how to design better algorithms utilizing these predictions.

We revisit the scalable hierarchical clustering algorithm designs explored in previous chapters and extend it to inputs in more general metric spaces. In Euclidean spaces we design cluster embeddings and couple it with ANN search to efficiently identify clusters to merge. However, the ANN technique is not known to exist for general metrics. We show how a **proxy metric**, which approximates the original metric, could be used to support the ANN search with minimal loss in hierarchical clustering performance.

Finally, we consider **correlation clustering**. Given a set of points along with recommendations whether each pair of points should be placed in the same cluster or into separate clusters, the goal is to cluster the points to minimize disagreements from the recommendations. We study this problem in the online setting, where points arrive one at a time, and upon arrival the algorithm must make an irrevocable cluster assignment decision. There is a simple lower bound that rules out any algorithm with a non-trivial competitive ratio. We propose using a small, randomized subset of nodes to help making online clustering decisions. Upon the arrival of a new node, the algorithm can check whether it is recommended to be in the

same/different cluster(s) with the reference set. We prove that the famous **Pivot** algorithm performs well in this setting. Moreover, the performance is robust to adversarial perturbations of the reference set.

Table of contents

1 Introduction 1

1.1 Theoretical Framework for Clustering Problems 2

1.1.1 Hierarchical Clustering: Objective Functions and Constraints 3

1.1.2 Faster Clustering: Scalable Average Linkage and Relational Algorithms 5

1.2 Improved Algorithm Design with Machine-Learned Predictions 6

2 A New Objective Function for Hierarchical Clustering 9

2.1 Introduction 9

2.2 Preliminaries 13

2.2.1 Objective Functions 13

2.2.2 Common Hierarchical Clustering Algorithms in Practice 14

2.3 Hierarchical-Split : Comparing Inter vs. Intra Cluster Distance 16

2.4 Ground-truth Inputs 17

2.4.1 De nition of Ground-Truth Inputs 18

2.4.2 Optimality of Generating Trees 19

2.5 Bisecting k-means Approximates Hierarchical-Split Objective 20

2.6 Randomly Partitioning Poorly Approximates the Hierarchical-Split Objective 25

2.7 Other Objectives for Data in Metric Space 29

2.8 Empirical Results 32

2.9 Conclusion 33

3 Fair Hierarchical Clustering 35

3.1 Introduction 35

3.2 Formulation 37

3.2.1 Generalized Objectives for Hierarchical Clustering 37

3.2.2 Notions of Fairness 38

3.3 Fairlet Decomposition 39

3.3.1 Fairlet decomposition for the value objective 40

3.4 Optimizing revenue with fairness 42
3.5 Optimizing value with fairness 43
3.6 Experiments 49
3.7 Conclusions 53

4 Scaling Average-Linkage via Sparse Cluster Embeddings 55
4.1 Introduction 55
4.2 Preliminaries 59
4.2.1 Notation System 60
4.2.2 Approximate Nearest Neighbors 60
4.3 Sparse Cluster Embeddings for Average Distance 62
4.4 Near Cluster Search 66
4.5 Fast Approximate Average-Linkage 67
4.5.1 A Sketch for ANN-based Average-Linkage 67
4.5.2 Robust Merging 68
4.6 Algorithm Analysis: Approximation Ratio and Running Time 70
4.6.1 Running Time Analysis for Algorithm 70
4.6.2 Approximation and Correctness 72
4.7 Experiments 75
4.8 Conclusions and Future Work 81

5 Relational Algorithms For K-Means Clustering 83
5.1 Introduction 83
5.2 Warm-up: Ef ciently Implementing 1-means++ and 2-means++ 89
5.2.1 Hardness of Relationally Computing the Weights 91
5.3 Related Work and Background 92
5.4 Simulating the k-means++ Algorithm 94
5.4.1 Relational Implementation of 3-means++ 94
5.4.2 Simulating k-means++ 97
5.5 Weighting the Centers 101
5.5.1 Analysis of the Weighting Algorithm 102
5.6 Conclusion 113

6 Scalable Hierarchical Clustering in General Metric Spaces 115
6.1 Introduction 115
6.2 Preliminaries 117
6.3 Warm-up: Using ANNs to Approximate Single Linkage 118

6.4 Main Algorithm for Scalable Average Linkage 120
6.5 Correctness and Run Time of Average Linkage Algorithm 123
6.6 Experiments 130

7 Robust Online Correlation Clustering 137
7.1 Introduction 137
7.2 Semi-Online Model for Correlation Clustering 140
7.3 Warm-Up: Pivoting Using a Random Sample 142
7.3.1 Pivot Preliminaries 142
7.3.2 Bounding Cost of $\mathcal{C}$ 143
7.3.3 Bounding Cost of $V \setminus S$ 144
7.4 Random Sample with Adversarial Corruptions 146
7.4.1 Analysis: Proof of Theorem 7.4.1 148
7.5 Experiments 153
7.6 Conclusion 156

References 159

Appendix A Background Information for Relational Database 169
A.1 Uniform Sampling From a Hypersphere 169
A.2 Background Information About Database Concepts 170

Appendix B Omitted Experimental Results for Hierarchical Clustering in General Metric Space 175
B.1 Performance And Running Time for Proxy-Hash-SL And Proxy-Hash-AL . 175
B.1.1 Robustness of Performance Against Sample Sizes And Parameter Tuning 176

Appendix C Supplementary for Semi-online Correlation Clustering 181
C.1 Experiments 181

Chapter 1

Introduction

Aiming at grouping similar objects together and separating dissimilar objects, clustering is one of the most common unsupervised learning problems. It has been widely used in areas such as pattern recognition and image analysis, where data mining tasks are common. In the past few decades many new clustering problems and algorithms have arisen in the community. This dissertation covers some of the most recent developments in clustering. It focuses on bridging the gap between clustering theory and practice by making them bene t from each other in two directions. We develop the theoretical foundation for clustering algorithms and uses the foundation to improve the state-of-the-art in practice. Similarly, we show how to rethink the combinatorial design of clustering algorithms by augmenting the algorithms with learned oracles using a data driven approach.

We rst look into different clustering problems and their state-of-the-art solutions through the lens of theory. It is common for ML/AI algorithmic solutions to be designed application-oriented, and become standard in practice despite an absence of a theoretical framework for their problem setting. Even very well-known methods, like Deep Learning, can be criticized for lacking theoretical support despite being prevalent among practitioners. Similarly, many clustering algorithms are also designed in lack of theoretical guidance. Some clustering problems could have complex structures, posing challenges for characterizing them mathematically. The arising needs in the ML/AI community, such as fairness and scalability, could also require updates in the current optimization framework. What are we optimizing and which solutions are good? How do we nd good solutions ef ciently? In this dissertation we answer these questions by building the theoretical foundation for clustering problems. This branch of work deepens our understanding of current algorithms, facilitates the communication of ML/AI ideas to the broad audience, and guides researchers in improving current algorithms and methods.

On the other hand, using ML oracles we can extract information from historical data which helps algorithms go beyond the worst case in traditional analysis. theory often designs algorithms using stylized models such as worst case analysis or stochastic models. However, these models are not a perfect characterization of practice. It would be ideal to have an algorithm tailored to the exact types of instances observed for the given application. Luckily, in practice there is usually lots of data available about past instances of the problem considered. One can learn information from this data and feed the information to an algorithm. The learned information contains predictions about data patterns in the current problem instance. The algorithm then uses the learned predictions to help make key decisions. In this way, the algorithm is tailored directly to instances seen. My work shows how to bound the running time and approximation ratios in terms of the errors in these predictions, giving a formal way to go beyond worst-case and stochastic models. The ultimate goal of this new approach for algorithm analysis is to discover new algorithmic solutions to our problems.

1.1 Theoretical Framework for Clustering Problems

The rst part of this dissertation develops a theoretical foundation that covers popular clustering problems/algorithms and their innovations. We start with formulating these problems as optimization frameworks, while incorporating new constraints characterizing requirements such as fairness. Guided by this framework we develop new solutions that beat state-of-the-art.

It is desirable to mathematically formulate learning problems in an optimization framework. Such a framework has an objective function characterizing the goal and a set of constraints characterizing the desired solutions. The design of such a framework may sometimes be self-explanatory. For example, for supervised learning problems knowing the true labels enables us to compute the prediction accuracy of learned models. Often we would aim at nding solutions that maximizes the accuracy, but with limited model complexity to avoid over tting. However, for unsupervised learning problems, where a “correct” solution might either not exist or be unknown to us, it can be a non-trivial task to nd a good optimization problem formulation.

For clustering problems like k-means, such problem formulations have been designed and well accepted by the community. However, for some other clustering problems, we have many algorithms that proved to be effective in practice, with no mature theoretical problem formulation. We would like to bridge this gap.

Goal 1: We develop well-formulated optimization framework for clustering problems.

Researchers can bene t from a well-designed optimization model in many ways. Its objective function gives a direct performance metric for comparing different solutions. The math formulation provides mathematical intuition on what solutions could be good. It is also the rst step towards analyzing algorithm performance from a theoretical point of view and developing algorithms with performance guarantees.

Goal 2: Equipped with the tool of optimization frameworks and algorithm analysis, we develop algorithms with both theoretical and empirical guarantees.

Fast, scalable algorithms are designed by applying combinatorial optimization and algorithm design tool boxes to the framework. Further, we test their performance and scalability on real datasets in various experiment settings. The algorithms beat the state-of-the-art, showing that theory can be used to guide practice.

1.1.1 Hierarchical Clustering: Objective Functions and Constraints

Hierarchical clustering is a core data analysis method. It takes a set of points and a pairwise similarity/dissimilarity score between the points as input. The output is a tree, often binary, whose leaves represent singleton clusters containing the points, and every internal node represents a cluster that is the union of the clusters represented by its children. The root is a cluster containing every point in the set.

One of the common complaints about k-means is that it is hard to choose the value of k in advance, which is the number of clusters. Hierarchical clustering does not require determining k a-priori. In fact, taking different layers of clusters in the tree gives clustering of different sizes and re nement levels. Although the tree structure of hierarchical clustering solutions eliminates such needs, it also makes it harder to design a framework capturing the characteristics of desirable solutions.

Fig. 1.1 A hierarchical clustering example. The left is input data set with visualized hierarchy of clusters. The right is the hierarchical clustering tree. The grey nodes are input points. Each internal node in the tree on the right is a cluster of the input points in its subtreem visualized using the same color on the left.

A new objective function. Chapter 2 designs a new objective function and explores its connection to practical algorithms for hierarchical clustering inputs in Euclidean space.

The results come from our paper [12]. In this setting, the input points have a pairwise dissimilarity score that is defined as the Euclidean distance. This setting is common for hierarchical clustering and is used in multiple hierarchical clustering ML packages (e.g., scipy and scikit-learn). Recent work [61], [115], [58] have proposed objective functions for different settings. However, they either do not directly apply to this setting, or do not make a significant distinction between good and bad solutions.

Our new objective function captures the key criterion in clustering: points in different clusters should have greater distances than points in the same cluster. This objective assigns penalizes the tree whenever it separates any pair of points into sub-clusters, whose distance is small compared to the average distance within the sub-clusters. It is also the first to show a theoretical connection between the proposed objective and the widely-used k-means, one popular hierarchical clustering algorithm used in practice. We complement our results by empirical studies that show the new objective function gives reasonable results for multiple standard algorithms in practice, showing it to be a reasonable performance measure.

Imposing fairness constraints. Chapter 3 studies fairness in hierarchical clustering, based on our work [14]. ML algorithms are often used in practice, where we need to have fairness constraints. Here we say a given algorithm is fair if its results are independent of certain sensitive features, such as traits of individuals which should not correlate with the outcome (e.g., gender, ethnicity, disability). This project seeks to build a new generation of ML algorithms that take into consideration these ethical and legal constraints.

We call such sensitive traits “protected features”. Removing the dependency of outcome on a protected feature is, in general, more non-trivial than simply dropping the feature from the input. This is due to possible correlation between protected and unprotected features. Even when a learning algorithm does not explicitly exploit protected features, it can still give biased results in retrospect. For example, when clustering demographic data we can get highly imbalanced clusters with regard to ethnicity, even if the clustering algorithm itself does not use this feature.

Fairness could be studied in various ways. We choose to characterize it mathematically and incorporate fairness constraints into clustering problems. How can we quantitatively define fairness and how should we design fairness constraints? Given the fairness constraints, does a fair solution exist and if so, how can we find it efficiently? How much do we lose in performance with respect to the objective function by imposing fairness constraints?

Chater 3 begins to answer these questions for hierarchical clustering. Given a protected feature with multiple possible values (called “colors”), we define fairness to be bottle-necked by the most dominant feature value. A data point set is called fair if none of its colors

take up an overly big portion of its points. We have designed algorithms for all prevalent hierarchical clustering objective functions proposed in recent work. Both theoretical analysis and empirical experiments show that fairness can be imposed with minimal loss in clustering quality.

1.1.2 Faster Clustering: Scalable Average Linkage and Relational Algorithms

In the following chapters the focus is shifted from developing mathematical frameworks to utilizing the framework to get algorithms that performs better in terms of scalability and/or solution quality. We seek algorithm designs that could speed up the execution, but can also provably perform well when using the objective function as the performance measure. That is, for any problem instances the output of the target algorithm should satisfy all constraints, as well as at least approximately optimizes the objective function.

The new algorithms are designed based on the theoretical foundation for these clustering problems. This dissertation complements the theoretical guarantees with empirical experiments whose settings cover some of the most common scenarios in practice. We show that the algorithms, being motivated and guided by the theoretical framework, beat the state-of-the-art clustering algorithms. Thus establishing the theoretical foundation of clustering problems could shed light on potential algorithmic innovations.

Scalable hierarchical clustering algorithms. Hierarchical clustering algorithms do not scale well to large data sets, as their running time grows at least quadratically with input size. Given input data in metric space with pairwise distance as a dissimilarity score, we consider agglomerative hierarchical clustering algorithms. These algorithms build the tree from bottom to top. They start with singleton clusters, and iteratively choose two clusters to merge that are most similar to each other. However, searching for closest clusters is time consuming.

Based on our work [100], Chapter 5 shows how to design a hierarchical clustering algorithm which has slightly super-linear running time in input size, breaking the previous quadratic running time bound. The key design here is to iteratively approximate the search for nearby clusters. Techniques are known to exist for searching for points with small distance in Euclidean space (namely, Approximate Nearest Neighbors (ANN)), but they do not directly apply to cluster searches. We design a cluster embedding which maps a collection of clusters to point sets in a Euclidean space. The mapped points are then used as surrogates for

clusters and coupled with ANN to speed up cluster searches. Our implementation empirically dominates the hierarchical clustering implementations in popular ML packages.

Clustering on relational inputs. New challenges on algorithm performance can also imposed by different data input formats. ML solutions traditionally assume a matrix input, where rows represent data points and the columns represent features. However, in practice data could be stored in different formats. One prevalent format is relational databases, a highly compact data type where features are stored only partially in each table and the underlying input could be found by concatenating the tables.

Current ML algorithm designs fall short of handling this new data format. The common practice is to convert the relational database into a matrix, which might cause the input size to grow exponentially. Naturally, the key questions arise: can we redesign current clustering algorithms so that they can skip the computationally expensive data transformation step and directly work on the relational databases? If so, which ML problems admit such solutions?

Chapter 5, which is based on our work [13], gives such an algorithm for the famous k-means clustering problem. The work adds to the fundamental understanding about the geometric properties of relational databases, and reveals the connection between relational database ML problems and combinatorial optimization.

1.2 Improved Algorithm Design with Machine-Learned Predictions

The second half of the dissertation focus on how the ML practice helps us approach algorithm design and analysis from a new perspective other than traditional worst-case analysis. With the aid of additional information provided by ML oracles, we go beyond previous running time and performance guarantees.

For every algorithm, there could be “hard” input instances where the algorithm struggles to find a good solution that would bottleneck standard worst-case analysis. The use of worst-case analysis is motivated by the assumption that the algorithm could be challenged by any adversarial input. The algorithm knows nothing about the input or optimal solution a-priori and always starts from scratch on a new input. However, in practice we seldom face the worst-case scenario. The worst-case inputs may not be a good characterization of the typical problem instances that we run into. Also, we have historical data about the input instances and the algorithms' outcomes. By applying data-mining approaches we could extract all kinds of knowledge that could be used to help an algorithm tailor towards the specific type

of problem instances in practice. The knowledge comes in the format of a Machine-learned oracle which is treated as a black-box by the algorithm, that gives information speci c to the possible problem inputs that we could have.

We call such an oracle a "predictor" . The de nition of a predictor can be very general and it depends on the problem settings. It could be a guess about a possibly good solution. If we are training a neural network, it is common practice to initialize the ML model with yesterday's training results, i.e., a "warm start", to make training more ef cient. Following the same logic, we can also initialize the algorithm's execution using the old results on another input instance. It could also some other intricate information about the input itself. For example, in an online problem setting where input arrives piece by piece, we can try to forecast the input to arrive in the future. Such a prediction is a natural t as it can help combat future uncertainty.

Goal 3: In various clustering problem contexts, nd the prediction that is most informative and design algorithms that bene t from these predictions. We have studied algorithm design with predictions for several clustering problems. In these settings we rst evaluate the prediction proposal and the new algorithm design from a theoretical point of view. We show that the predictions indeed help us design algorithms that go beyond previous performance guarantees. We then empirically show that such predictions can be obtained in real life, and even with moderate accuracy, can be bene cial to algorithm results.

Scalable hierarchical clustering algorithms with proxy metric. In Chapter 6, we revisit the problem of designing scalable agglomerative hierarchical clustering algorithms in a general metric space setting. See our paper [114] for more details.

In Chapter 4 we consider points in Euclidean space, where ANN techniques are known to exist and are used to couple to other algorithm designs to speed up the search for nearby clusters. Many problems, however, are best captured by general metric spaces for which ANN schemes either do not exist, or may be expensive to compute. In this case, we might be able to discover a proxy metrics de ned on input points, which approximates the original metric but supports ANN schemes.

Such proxies could be obtained in many different ways. It might arise naturally from the problem de nition. For example, on a road network, consider the shortest path distance and the geographic distance. These two metrics, despite not being equivalent, could be good surrogates of each other. One can also get a proxy metric from more sophisticated ML techniques, such as applying dimensional reduction approaches to high-dimensional metrics. The theoretical arguments in Chapter 6 are complemented by empirical evaluations that closely examine different general/proxy metric settings.

Online correlation clustering with predictions. Correlation Clustering is a graph-based clustering method often used to aggregate inconsistent information. Given a set of points along with recommendations whether each pair of points should be placed in the same cluster or into separate clusters, the goal is to cluster the points to minimize disagreements from the recommendations. In [93], we study the correlation clustering problem in the online setting, where points arrive one at a time, and upon arrival the algorithm must make an irrevocable cluster assignment decision.

While the online version is natural, the problem is hard as seeing only partial information could mislead the algorithm to make wrong decisions that result in many disagreements later on. See Figure 1.2 for an illustration. Indeed, there is a simple lower bound that rules out any algorithm with a non-trivial competitive ratio.

Fig. 1.2 A hard online correlation clustering example. The blue lines represent same-cluster membership recommendations. The subgraph with the blue solid lines in the middle arrive first, misleading an online algorithm to put them together. However, the true underlying graph might also contain the blue dashed lines so the optimal solution should separate them.

Our proposed prediction is a small set of nodes called the *advice set*. Upon the arrival of a new node, we are allowed to refer to the advice set, and make cluster assignments based on the same/different cluster recommendations between the new node and the advice set. Such nodes and the recommendations could be obtained by learning from historical data. For example, if the underlying graph represents a social network where nodes are entities and edges are positive/negative communication, one can learn such predictions from last week's records. Empirically we test these settings and show great improvement in performance when introducing the advice set.

Chapter 2

A New Objective Function for Hierarchical Clustering

2.1 Introduction

In this chapter we focus on hierarchical clustering as introduced in Chapter 1. Recall that this problem takes as input a set of points with a pairwise score that represents the similarity/dissimilarity, and outputs a (binary) tree whose internal nodes represent a hierarchy of clusters. The leaves of this tree correspond to the individual input data points, and each internal node is a cluster of the leaves in the subtree rooted at it. When a node gets closer towards the leaves, the cluster it represents should become more re ned, and the points in this cluster should become more similar to each other. The nodes of the same level in this tree represent a partition of the given data set into clusters. Note that each data point (leaf) belongs to many clusters, one for each ancestor. Figure 2.1 illustrates an example application of hierarchical clustering: biological taxonomy. The data points could represent different species and the internal nodes are different levels of categories. From bottom to top, the point cat could belong to the clusters representing mammals, animals, life, etc.

Fig. 2.1 Another illustration of a hierarchical clustering tree.

The mainstream algorithms used to do hierarchical clustering can be classi ed roughly into two categories: agglomerative and divisive, depending on whether the construction

happens from bottom to top or top to bottom. Agglomerative algorithms initialize every point to be a singleton cluster. They iteratively pick the two clusters that are the most similar to each other to merge into a bigger cluster until only one cluster remains. Meanwhile, the algorithms create a parent in the hierarchical tree produced that is connected to the two nodes corresponding to the two clusters before merging. Divisive algorithms, on the other hand, initialize the whole point set as one single cluster, and create a root node corresponding to this cluster in the hierarchical tree. They iteratively split a cluster into smaller clusters. Then the algorithms create nodes representing the separated clusters in the tree and make them the children of the parent node. Again, the notion of similarity is open to different interpretations. A divisive algorithm terminates when every point is in its own individual cluster. Data scientists regularly use the bisecting k-means algorithm at each level.[1] This is used when the distances between the data points are used as dissimilarity scores. Existing work [9] proved that popular divisive algorithms can produce clusterings different from agglomerative algorithms. Naturally, they may be optimizing fundamentally different criteria.

Recently the community has developed an interest in identifying a natural global objective function for hierarchical clustering. However, existing work has had more success connecting these objective functions to agglomerative rather than divisive algorithms. Dasgupta [61] developed a cost function objective for data sets with similarity scores between points. The work has initiated an exciting line of study [14, 58, 45, 46, 68]. Cohen-Addad et al. [58] generalized the results in Dasgupta [61] into a class of cost functions that possess properties desirable of a valid objective function. They showed that average-linkage, one of the most popular agglomerative algorithms, is a $\frac{2}{3}$ approximation for an objective which is based on the Dasgupta [61] objective but handles dissmilarity scores.[2] Contemporaneously, Moseley and Wang [115] designed a revenue objective function based on Dasgupta [61] and showed average-linkage is a constant approximation for the objective. Charikar et al. [45] showed an improved analysis of average-linkage for Euclidean data. Together Cohen-Addad et al. [58], Charikar et al. [45] and Moseley and Wang [115] have established a relationship between a practically popular algorithm and global objectives.

This chapter is interested in the following aspects of objective function designs and their connection to popular algorithms in practice.

Euclidean data. We consider data embedded in Euclidean space where the ℓ_2 distance between points represents their dissimilarity. There is currently only one global objective, the objective of Cohen-Addad et al. [58], that has been proposed for data with dissimilarity

[1] The word "bisecting" refers to the case when $k = 2$.

[2] Throughout this paper we use $c > 1$ for approximations on minimization problems and $c < 1$ for maximization.

scores. Noticeably, all trees gave a constant approximation if the data is in a metric space [58]. Here we give a stronger conclusion and show in Section 2.7 that every tree is a $\frac{1}{2}$-approximation on metric input. Therefore, in a Euclidean space, the objective does not make a large differentiation between different clusterings. Is there a natural objective that makes a stronger distinction between good and bad solutions? This is the target question this paper addresses.

Divisive algorithms. While great strides have been made on the foundations of hierarchical clustering, it remains an open question to explain what popular divisive algorithms optimize. In particular, the popular bisecting k-means algorithm has been proven to be at least a factor $O(\sqrt{n})$ far from optimal for the objectives given in Moseley and Wang [115] and Dasgupta [61]. This can be viewed as these algorithms being extremely bad for these objectives in the worst case. This contrasts with the performance of average-linkage for known objectives. Perhaps, this highlights that bisecting k-means and other divisive algorithms optimize something fundamentally different than average-linkage and general linkage based algorithms. It remains to discover a global objective that helps characterize the optimization criteria of divisive algorithms, another target of this paper.

Our contributions. This paper introduces a new revenue maximization objective for hierarchical clustering on a point set in Euclidean space. The objective is designed to capture the main criterion that motivates the use of divisive algorithms: when data is split at a level of the tree, the data in each sub-cluster should be closer to each other than data points in different clusters.

Each node in the tree corresponds to a split that generates revenue. The objective specifies that the global revenue of the tree is the summation of the revenue at each node. The split revenue captures the quality of the split.

Guiding Principle: The new objective function enforces that a split is good if the inter-cluster distances are big compared to intra-cluster distances[3], as is indicated in Figure 2.2. This is the main motivation behind a generic divisive algorithm. Of course, the global tree structure influences the possible revenue at an individual split.

We show several interesting properties of this new objective.

- For problem instances corresponding a ground-truth as introduced in Cohen-Addad et al. [58], this objective gives desirable optimal solutions. In particular, Cohen-Addad et al. [58] introduced a large class of instances that have a natural corresponding

[3] Here "inter-cluster distances" refers to that between points in different clusters, while "intra-cluster distances" refers to that between points in the same cluster.

Fig. 2.2 Intra- and inter- cluster distance of two clusters. The black pair is an example of intra-cluster pairs, and the grey pair is an example of inter-cluster pairs.

hierarchical clustering that should be optimal. We prove that these trees are optimal for the new objective function we propose on such instances. We note that these instances generalize instances given in Dasgupta [61] that were used to motivate a hierarchical clustering objective.

- The bisecting k-means algorithm is a constant approximation for the objective. This establishes that the objective is closely related to the bisecting k-means algorithm and aids in understanding the underlying structure of solutions the algorithm produces. This is the first global objective that this algorithm is known to provably optimize.
- The objective is trivially modular over the splits, like all three previous objectives.
- In the context of metric spaces, this objective has different properties compared to some proposed objectives. It is known that the Random algorithm[4], which partitions data uniformly at random at each node, is a constant approximation for both objectives in [58] and [115]. For these two objectives, Random is a $\frac{2}{3}$ and $\frac{1}{3}$ approximation, respectively. The Random algorithm can produce undesirable hierarchical clusterings and it is counterintuitive that it should be a constant approximation for these objectives. This paper shows that Random results in a $O(\frac{1}{n^{\epsilon}})$-approximation for the proposed objective for a constant $\epsilon > 0$. Therefore, Random provably performs poorly for the new objective.

We further show the following about other objectives in metric space. These show that some other objectives do not make a large differentiation between trees in metric space, even if the trees correspond to a poor clustering. Our objective does and this can be seen as an advantage of the new objective.

- As mentioned, we show that every tree is a $\frac{1}{2}$-approximation for value objective when points have dissimilarity scores that form a metric.
- We show that every tree is a 2-approximation for the Dasgupta objective [61] for similarity scores that satisfy the triangle inequality. We include this result to provide

[4] See Section 2.6 for a formal description of the algorithm.

insight into this objective. However we note that this is less surprising than the similar result on the Cohen-Addad objective [58] since some natural similarity score instances do not satisfy the triangle inequality.

We investigate empirically the performance of three popular algorithms used in practice and Random algorithm for the new objective. As is suggested by theory, the proposed objective moderately favors bisecting k-means over two agglomerative algorithms, while magnifying the gap between the performance of Random and the other three algorithms.

Other related work. Some work discusses about different hierarchical clustering algorithms and their performance. Steinbach et al. [129], Murtagh [117], Murtagh and Contreras [118] and Zhao et al. [138] discussed about common agglomerative algorithms and compared their performance in a variety of backgrounds. Ackerman and Ben-David [9] investigated properties of trees produced by linkage-based agglomerative algorithms. See Steinbach et al. [129] and Murtagh and Contreras [118] for more information on divisive algorithms.

Other work centers around when bisecting algorithms work well. The work of Dasgupta and Long [63], Plaxton [120] show the remarkable result that hierarchical trees exists such that each level of the tree optimizes the corresponding k-clustering objective. These algorithms are complex and are mostly of theoretical interest. Balcan et al. [28] showed that partitioned clusterings can be uncovered from using hierarchical clustering methods under stability conditions. The work of Awasthi et al. [24], Balcan et al. [27], Carlsson and Mémoli [38] and pointers therein study stability conditions of clustering.

2.2 Preliminaries

This section includes the preliminaries for hierarchical clustering that will be adopted in all chapters concerning this topic (including Chapter 3, 4 and 6).

2.2.1 Objective Functions

For a given data set V with n points, let $s(i, j)$ be the similarity score between points i and j if similarity is used, or $d(i, j)$ if distance/dissmilarity score is used instead. One can also think of this input as a complete graph with weighted edges $G = (V, s)$ or (V, d). Without loss of generality, in this chapter we assume $V = \{1, 2, \ldots, n\} = [n]$.

In a tree T, let $[i _ j]$ denote the Least Common Ancestor of i and j. Let $T[u]$ denote the subtree in T rooted at u. Then, $T[i _ j]$ denote the subtree rooted at i and j's least common ancestor. We let $\text{leaves}(T[i _ j])$ denote the set of leaves of $T[i _ j]$.

There are three objective functions for hierarchical clustering in different settings. We refer to them as *cost*, *value* and *revenue*, respectively.

Cost. The original objective, cost, is introduced by Dasgupta [6] for inputs based on similarity scores $s(\cdot,\cdot)$. This is a cost function which a hierarchical clustering tree should seek to minimize.

Definition 2.2.1 (Cost). The *cost* of a tree T for an instance $G = (V, s)$ where $s(\cdot,\cdot)$ denotes similarity is: $cost_G(T) = \sum_{i,j \in V} s(i,j) \cdot |leaves(T[i \lor j])|$.

Since every $s(i,j)$ is multiplied with the number of leaves of the smallest tree containing both i and j, the points that are more similar (bigger $s(i,j)$'s) are encouraged to have $T[i \lor j]$ with fewer leaves. In other words, from bottom to top, these points should be merged into the same cluster before other points that are less similar.

Value. The motivation in value, proposed by Yu [15] is similar to cost, except now the similarity score $s(i,j)$ is swapped to a dissimilarity score $d(i,j)$ and the problem is changed to a maximization problem.

Definition 2.2.2 (Value). The *value* (*val*) of a tree T for an instance $G = (V, d)$ where $d(\cdot,\cdot)$ denotes distance is: $val_G(T) = \sum_{i,j \in V} d(i,j) \cdot |leaves(T[i \lor j])|$.

Revenue. Finally, Moseley and Wang [11] introduced the revenue objective for hierarchical clustering on similarity-based input. Also note that for a given input graph $Graph(V, s)$, the cost and revenue objectives sum up to a constant $|V| \cdot \sum_{i,j \in V} s(i,j)$.

Definition 2.2.3 (Revenue). The *revenue* (*rev*) of a tree T for an instance $G = (V, s)$, where $s(\cdot,\cdot)$ denotes similarity between data points, is $rev_G(T) = \sum_{i,j \in V} s(i,j) \cdot (|V| - |leaves(T[i \lor j])|)$.

Throughout this dissertation, we focus on hierarchical clustering inputs with dissimilarity score $d(\cdot,\cdot)$. Additionally in this chapter we assume that V lies in a Euclidean space, using $d(i,j) = \|i - j\|_2$ which is the ℓ_2 norm as the distance/dissimilarity score. We note that only the *value* objective applies to this setting. In the following sections we will show value has several undesirable properties.

2.2.2 Common Hierarchical Clustering Algorithms in Practice

Agglomerative Algorithms. Depending on whether the input adopts similarity/dissimilarity scores, an agglomerative algorithm quantifies cluster similarity/dissimilarity and iteratively choose clusters to merge with highest similarity, or lowest dissimilarity. Cluster similarity can be defined in different ways which lead to different agglomerative algorithms. Given two

sets A and B and a score $x(\cdot,\cdot)$ between points $x \in \{s, d, g\}$, here is a list of agglomerative algorithms and similarity/dissimilarity measures between clusters:

- Average-linkage. The cluster measure is $Avg(A,B) = \frac{\sum_{i \in A, j \in B} x(i,j)}{|A||B|}$ for both similarity and dissimilarity. Average-linkage seeks to maximize this when s is used and minimize when d is used.
- Single-linkage. This method works on inputs with distance metrics. The cluster measure is $d(A,B) = \min_{i \in A, j \in B} d(i,j)$.
- Centroid-linkage. This works on inputs in Euclidean space equipped with distance metrics used as dissimilarity scores $d(A,B) = d(m(A), m(B))$ where $m(\cdot)$ is the centroid of a set.

Together these three algorithms are called linkage-based algorithms.

Divisive Algorithms. On the other hand, divisive algorithms seek to separate a cluster into sub-clusters such that the points in different sub-clusters are as dissimilar/distant from each other as possible. The most used divisive algorithm is bisecting k-means, which finds the sub-clusters by optimizing k-means objective.

We first define the k-means objective. Given a point set S, a k-means clustering partitions S into k sets $S_1, S_2, \ldots, S_k$. The k-means objective calculates the summation over the squared norm of the distance between a point to the centroid of the set it belongs to: $\sum_{j=1}^{k} \sum_{u \in S_j} d^2(u, m(S_j))$. Here again $m(S_j)$ denotes the centroid of S_j.

Let $D_k(S)$ denote the optimal k-means objective function value for the point set S, where k is the number of clusters. We will be particularly interested in $D_2(S)$, the optimal 2-means objective.

Fix a hierarchical clustering tree T on a set V. Consider a node of the tree and let V be the subset of input data that is input to the current split. These will eventually be the leaves of the subtree induced by this node. We use (S_1, S_2) to denote a split in the tree where a set S is separated into two non-empty subsets S_1 and S_2. These sets correspond to the input of the two child nodes. We let $S \to (S_1, S_2) \in T$ denote that this split exists in T.

Any split $S \to (S_1, S_2)$ where S_1 and S_2 are a partition of S is a valid 2-means solution for the point set S. Since $D_2(S)$ denotes the optimal objective function value, $D_2(S) \le D_1(S_1) + D_1(S_2)$ by definition of the 2-means objective. In particular, when using bisecting k-means since $S \to (S_1, S_2)$ is the optimal 2-means solution, we have $D_2(S) = D_1(S_1) + D_1(S_2)$.

2.3 Hierarchical-Split : Comparing Inter vs. Intra Cluster Distance

This section de nes the new objective which is a revenue function. We call the problem of optimizing this objective the **Hierarchical-Split problem** . We de ne a revenue for each pair of input points depending on where they got seperated from each other in the tree. The objective function is then the aggregate of this revenue over all pairs of points.

De ning pairwise revenue. Consider a node in a hierarchical clustering representing a set S and this set is split into S_1 and S_2. A good tree ensures that the pairs of points $i, i' \in S_1$ (resp. S_2) are more similar that pairs $i \in S_1$ and $j \in S_2$ (i.e. $d(i,j) \geq d(i,i')$). This ensures the points corresponding to the cluster at a node in the tree become more similar at lower levels of the tree. We say i and j are *split* the first time they no longer belong to the same cluster from top to bottom.

Every pair i and j will be eventually split in the tree and a hierarchical clustering objective should ensure they are split at the appropriate place in the tree. Further, an objective should optimize over all pairs uniformly to determine the splits.

Guided by these principles, we develop the objective as follows. We begin by allowing every pair i and j to generate at most one unit of revenue. When the pair is split at an appropriate position in the tree more revenue will be generated, and less revenue (or even 0) will be obtained when the pair is separated at a poor position. The maximum, one unit of revenue, can always be obtained for a fixed pair, but not necessarily for all pairs simultaneously. This is the key to determine the quality of a split.

Say that $S \to (S_1, S_2)$ is the split at some node in the tree and $i \in S_1$ and $j \in S_2$ are split. As discussed above, points in S_1 (respectively S_2) should be more similar to each other than i and j. To measure the similarity of i to other points in S_1 we use $d(i, m(S_1))$, the distance of i to the centroid of S_1. Similarly, we use $d(j, m(S_2))$ to measure the distance of j to points in S_2. The distance of a point to the centroid of a set measures the distance to the average point in the set. Thus, we would like $d(i,j)$ to be larger than both $d(i, m(S_1))$ and $d(j, m(S_2))$ for it to make sense to split i and j. That is, i and j should become more similar to their respective sets after the split than they are to each other.

Formally, de ne the revenue for a pair of points as follows. Let $d_{S_1,S_2}(i,j) = \max\{d(i, m(S_1)), d(j, m(S_2))\}$ be the maximum distance of i and j to their respective centroids. We would like $d_{S_1,S_2}(i,j)$ to be smaller than $d(i,j)$ and therefore i and j generate a unit of revenue when this is the case. When $d_{S_1,S_2}(i,j) \geq d(i,j)$ we assume the revenue decays linearly. That is, the revenue is $\frac{d(i,j)}{d_{S_1,S_2}(i,j)}$.

Putting the above together, define the revenue for splitting i and j as $rev(i,j) = \min\{\frac{d(i,j)}{d_{S_1,S_2}(i,j)}, 1\}$[5]. This is the revenue i and j generates. Again, a revenue of a unit can always be obtained since we can let j be the unique last point split from i. However, a good overall hierarchical splitting structure is needed to get good revenue for many pairs of points, thus obtaining global performance guarantees for the tree.

The global objective. The global objective is defined as follows.

Definition 2.3.1 (Hierarchical-Split) For a data set V and a given hierarchical clustering tree T, define the hierarchical tree revenue function as follows. Let $rev(S_1,S_2) = \sum_{i\in S_1}\sum_{j\in S_2} rev(i,j)$ $= \sum_{i\in S_1}\sum_{j\in S_2}\min\{\frac{d(i,j)}{d_{S_1,S_2}(i,j)}, 1\}$ be the revenue over all pairs of points split across S_1 and S_2. The aggregate revenue is $rev_T(V) = \sum_{\{i,j\}\subseteq V} rev(i,j) = \sum_{S_1,S_2):(S_1,S_2)\in T} rev(S_1,S_2)$, and it should be maximized over all trees.

As is shown in Definition 2.3.1, there are two ways of computing $rev_T(V)$. One is to sum up the revenue over the pairs, while the other is to sum up the revenue over the splits. Both methods lead to the same value. The second form allows us to judge whether a split at some internal node of the tree is good or not compared to the number of pairs it separated.

2.4 Ground-truth Inputs

The work of Cohen-Addad et al. [58] gave a characterization of desirable hierarchical clustering objectives. The idea is to give a class of instances that naturally correspond to a specific hierarchical clustering tree. These trees should be optimal solutions for a good hierarchical clustering objective.

In particular, Cohen-Addad et al. [58] defined input instances that correspond to ultrametrics. Such inputs will be referred to as *ground-truth* inputs. For such an input, they define *generating trees*, which should be optimal for the hierarchical clustering objective to be valid. Intuitively, in an ultrametric either it is clear what the split should be at each point in the tree or all splits are equivalent[6]. The resulting tree is a generating tree.

We prove a generating tree is indeed an optimal solution for our objective, if the input in Euclidean space is ground-truth. This perhaps shows that optimizing Hierarchical-Split could give desirable results.

[5] We assume dividing by 0 gives revenue 1.

[6] If there is a natural split then the points can be divided into two groups A and B such that inter-group distances are larger than intra-group distances. If all splits are equivalent then pairwise the points are all the same distance.

2.4.1 De nition of Ground-Truth Inputs

We cite the following de nitions from Cohen-Addad et al. [58].

De nition 2.4.1. A metric space$(X; d)$ is an ultrametric if for every$x; y; z 2 X$, $d(x; y)$ $\max f d(x; z); d(y; z)g$.

Intuitively, the de nition of ultrametric implies that any three points$u; v; w$ form an isosceles triangle, whose equal sides are at least as large as the other side. Cohen-Addad et al. [58] then de ned an instance generated from ultrametric, which is treated as ground-truth input for hierarchical clustering.

De nition 2.4.2. An input instance on a set of points$V$ with pairwise distance function$d$ is generated from an ultrametric if the distances function$d$ corresponds to a ultrametric.

Following Cohen-Addad et al. [58], we de ne generating trees on distance-based input, which are considered the most well-behaving hierarchical clustering trees for a ground-truth input.

De nition 2.4.3. If the instance$V$ is generated by ultrametric, a binary tree$T$ is a generating tree for$G$ if it satis es the following properties:

1. It hasjVj leaves andjVj 1 internal nodes. Let$L$ denote its leaves and each point$L$ in corresponds to a unique point$v$ in Let$N$ denote its internal nodes, corresponding to clusters of the leaves of the subtree rooted at the node.

2. There exists a weight function$W : N$ 7! R_+. For$N_1; N_2 2 N$, if N_1 is on the path from N_2 to the root,$W(N_1)$ $W(N_2)$. For every$x; y 2 V$, $d(x; y) = W(LCA_T(x; y))$, where$LCA_T(x; y)$ denotes the Least Common Ancestor of leaves corresponding to and$y$ in T.

Cohen-Addad et al. [58] proposed that for a ground-truth input graph as de ned in De nition 2.4.2, if there exists any corresponding generating tree$T$, it is considered one of the best solutions among all the solutions, and thus should be one of the optimal solutions for the hierarchical clustering objective function used.

We give some intuition for why a generating tree is considered the best tree on such inputs. The value$W(LCA_T(x; y))$ can be interpreted as the distances of edges cut in split at the LCA of$x$ and$y$. The higher up this LCA is, the larger the split distance should be. All points split at the LCA node have equal pairwise distance, later we will show that this is exactly the maximum pairwise distances in the corresponding point set. This agrees with the driving motivation for the tree design as a split should always separate the farthest points. The next subsection articulates these desirable properties.

2.4.2 Optimality of Generating Trees

Now we prove that given an input that is generated from an ultrametric, every generating tree is optimal for Hierarchical-Split function introduced in this paper. In particular, every pair of points will get full revenue.

Lemma 2.4.1. A binary tree T, with $|V|$ leaves corresponding to the points V and $|V| - 1$ internal nodes, is a generating tree for an instance V generated from a ultrametric if and only if it satisfies the following property:

- For every split $A \cup B \to (A, B)$ from top to bottom, $\forall i \in A, j \in B, d(i, j) = \max_{x \in A, y \in B} d(x, y)$.

Proof. The if direction is true, since from top to bottom, at each split the Tree cuts only the longest distances in the current set of points, both properties in Definition 2.4.3 trivially hold. We prove the only if direction. For every $i \in A$ and $j \in B$, $LCA_T(i, j)$ is always the node representing $A \cup B$, so $d(i, j) = W(LCA_T(i, j))$ is always the same value. To show it is the maximum distance in all the pairwise distances in $A \cup B$, assume that is not the case. Then some pair of vertices of maximum distance is contained in the subgraph induced by A or B, which means it will be cut in the subtree rooted at A or B, say it is cut at root N_2, and let $N_1 = LCA_T(i, j)$. N_1 is on the way from N_2 to the path, but $W(N_1) < W(N_2)$, contradicting property (1). □

Every ground-truth input has at least one generating tree, as stated in the following theorem.

Theorem 2.4.2. For every instance generated from some ultrametric, there is always a generating tree T as defined in Definition 2.4.3.

Proof. We prove the theorem by constructing one such tree in the following way. Say given a set S, we separate it into two sets L and R.

1. Pick a pair of points (i, j) with longest distance. Put i into L and j into R.
2. For any point $x \in S$, either $d(i, x) < d(i, j)$ or $d(i, x) = d(i, j)$ since $d(i, j)$ is chosen to be the maximum. If $d(i, x) = d(i, j)$ put x into R, otherwise put it into L.

To argue that all points in L and R are of distance $d(i, j)$ from each other, notice that if L only contains x the theorem trivially holds. Otherwise, apparently by construction we also have $\forall y \in R, d(i, y) = d(i, j)$. Now take any two points $x \in L, y \in R$, we further prove $d(x, y) = d(i, j)$. Observe that $d(i, x) < d(i, j)$ but $d(i, y) = d(i, j)$. Again by definition of ultrametric, in the triangle formed by i, x, y, we have $d(x, y) = d(i, y) > d(i, x)$. By Lemma 2.4.1, this is a generating tree for G. □

Using Lemma 2.4.1, the optimality of T is proved by arguing every split gives a revenue of 1 for every pair of points it separates.

Theorem 2.4.3. *A generating tree T for an instance generated from an ultrametric is optimal for the Hierarchical-Split objective.*

Proof. Given any split in the tree $A \cup B \to (A, B)$, for any $i \in A$ and $j \in B$, we prove that $d(i, m(A)) \leq d(i, j)$ and $d(j, m(B)) \leq d(i, j)$. As a result, $rev(i, j) = \frac{d(i,j)}{\max\{d(i,m(A)), d(j,m(B)), d(i,j)\}} = 1$.

Let's focus on A for the time being. By Lemma 2.4.1, $\forall x \in A, d(i, x) \leq d(i, j)$. By convexity of norms, $d(i, m(A)) = d(i, \frac{\sum_{x \in A} x}{|A|}) \leq \frac{\sum_{x \in A} d(x,i)}{|A|} \leq d(i, j)$. The other inequality, $d(j, m(A)) \leq d(i, j)$, can be proved in the same way. □

2.5 Bisecting k-means Approximates Hierarchical-Split Objective

This section shows that the bisecting k-means algorithm is a constant approximation for the proposed objective. This establishes a foundational connection between a natural objective function and the bisecting k-means algorithm. This is the first analysis showing that bisecting k-means optimizes a global objective function. This helps explain the structure of the solutions produced by the algorithm.

The goal of this section is to show the following theorem.

Theorem 2.5.1. *Fix any input set V and let T be the tree created by the bisecting k-means algorithm. The tree T is a constant approximation for the Hierarchical-Split objective.*

The analysis is based on analyzing each split performed by bisecting k-means individually. The following lemma shows that if every split in a hierarchical clustering tree is good for the objective function proposed, then the whole tree is also good. By "good" we mean that the split gains a revenue which is at least some constant factor times the number of pairs separated. This lemma follows immediately by definition of the objective.

Lemma 2.5.2. *A hierarchical clustering tree T is a g-approximation for the Hierarchical-Split problem if it satisfies the following condition: $\forall S \to (S_1, S_2) \in T, rev(S_1, S_2) \geq g|S_1||S_2|$ holds for some constant $g > 0$.*

The above lemma allows us to focus on a single iteration of the bisecting k-means algorithm. Suppose at some iteration, a cluster B is split into A and B. We give the following definition of a high-revenue point: A point u in A is a high-revenue point if for most of the points in B, it gains acceptable amount of revenue.

De nition 2.5.1. Given a split $A [B$ and a point $u 2 A$, we de ne high-revenue set in B for u as: $HR_B(u) = f v 2 B : rev(u;v) \frac{1}{10} g$. De ne the low-revenue set for $u 2 A$ in B as: $LR_B(u) = f v 2 B : rev(u;v) < \frac{1}{10} g = B n HR_B(u)$.

De nition 2.5.2. Given a split $A [B$, a point $u 2 A$ is a high-revenue point if $jHR_B(u)j \frac{1}{2}jBj$. Otherwise, it is called a low-revenue point.

With the de nition of high-revenue points in place, the next lemma claims that given split $A [B ! (A;B)$ created by the optimal 2-means algorithm, if $jAj j Bj$, at least half of A are high-revenue points. This is the main technical lemma. This combined with Lemma 2.5.2 implies Theorem 2.5.1.

Lemma 2.5.3. Let A and B be the optimal 2-means solution for the point set $A [B$. Without loss of generality, suppose $jAj j Bj$. Then, at least $\frac{4}{7}jAj$ points in A are high-revenue. This gives a lower bound of at least $\frac{1}{35}jAjjBj$ revenue in total for splitting A and B.

The rest of the section is devoted to proving Lemma 2.5.3 by contradiction, with proofs partially omitted due to space limits. For the rest of the section x a set A and B and let the partition $A;B$ correspond to the optimal solution to the 2-means problem on $A [B$. For sake of contradiction suppose more than $\frac{3}{7}jAj$ points in A are low-revenue points. We will show that such a split $A [B ! (A;B)$ cannot be optimal for the 2-means objective. Indeed, we will show that another split has a smaller 2-means objective value, proving the lemma.

Say we have $i 2 A$ and $j 2 B$, such that $rev(i;j) < \frac{1}{10}$. Let H be the hyperplane such that $H = f y : d(y;m(A)) = d(y;m(B)) g$. Then, H separates the Euclidean space into two half-spaces $H^+ = f y : d(y;m(A)) d(y;m(B)) g$ and $H = f y : d(y;m(A)) d(y;m(B)) g$. By the assumption that the split $A [B ! (A;B)$ is the optimal 2-means solution, we have $A H^+$, and $B H$. Next we show the following structural lemma. This lemma says that if $rev(i;j)$ is small then $d(i;m(A))$ and $d(j;m(B))$ are within a constant factor of each other, which is close to 1. Geometrically, this implies that both i and j are located close to the hyperplane H. See Figure 2.3 for an illustration. The following lemma's proof is in the appendix.

Lemma 2.5.4. Consider any $i 2 A$ and $j 2 B$. If $rev(i;j) < \frac{1}{10}$, we have $\frac{9}{10}d(i;m(A)) < d(j;m(B)) < \frac{10}{9}d(i;m(A))$, and $\frac{9}{10}d(j;m(B)) < d(i;m(A)) < \frac{10}{9}d(j;m(B))$, and $d(i;j) < \frac{1}{9}\min f d(i;m(A)) ; d(j;m(B)) g$.

Proof. Say that $rev(i;j) < \frac{1}{10}$. Without loss of generality assume that $d(i;m(A)) d(j;m(B))$. This and the de nition of revenue give $d(i;j) < \frac{1}{10}d(i;m(A))$. Since A and B is the optimal 2-means partition, $d(i;m(A)) d(i;m(B))$ and $d(j;m(B)) d(j;m(A))$. The triangle inequality

Fig. 2.3 Proof by constructing a better 2-means solution. The bold dashed line in the middle is the hyperplane H. The two bold ellipses are clusters A and B, respectively. The dashed ellipse in A is the set S, and the dashed ellipse in B is the low-revenue set $LR_B(u)$ for point $u \in A$ in B. S and $LR_B(u)$ are both close to the separating hyperplane H. A new partition $A \cup B \to (A \setminus S, B \cup S)$ is constructed, represented by the two grey areas.

gives,

$$\begin{aligned} d(j, m(B)) \quad & d(i, m(B)) \quad d(i, j) \quad d(i, m(A)) \quad d(i, j) \\ > \; & d(i, m(A)) \quad \frac{1}{10} d(i, m(A)) = \frac{9}{10} d(i, m(A)) \end{aligned}$$

An analogous proof shows $d(i, m(A)) > \frac{9}{10} d(j, m(B))$. The last inequality in the lemma follows immediately from these two inequalities. □

Let S be the subset of low-revenue points in A. By assumption $|S| > \frac{3}{7}|A|$. The next lemma establishes that any two points in S are very close to each other as compared to their distance to the centroid $m(A)$. The following lemma's proof is in the appendix.

Lemma 2.5.5. Let S be the low-revenue points in A. For any two points $u, v \in S$, $d(u, v) \leq \frac{2}{9} \max\{d(u, m(A)), d(v, m(A))\}$.

Proof. Recall that $LR_B(u)$ is the set of points in $w \in B$ such that $rev(u, w) < \frac{1}{10}$. Similarly for $LR_B(v)$. Knowing that $|LR_B(u)| > \frac{1}{2}|B|$ and $|LR_B(v)| > \frac{1}{2}|B|$, there exists some point $w \in B$, such that $rev(u, w) < \frac{1}{10}$ and $rev(v, w) < \frac{1}{10}$. Without loss of generality suppose $d(u, m(A)) \geq d(v, m(A))$. We want to show $d(u, v) \leq \frac{2}{9} d(u, m(A))$, notice that $d(u, v) \leq d(u, w) + d(v, w)$, and we have $d(u, w) \leq \frac{1}{9} d(u, m(A))$ and $d(v, w) \leq \frac{1}{9} d(v, m(A))$, respectively, by Lemma 2.5.4. Note that $d(v, m(A)) \leq d(u, m(A))$, so $d(v, w) \leq \frac{1}{9} d(u, m(A))$, and we conclude that $d(u, v) \leq \frac{2}{9} d(u, m(A))$. □

Let x be the point in S such that $x \in \arg\max_{u \in S} d(u, m(A))$, the farthest points from $m(A)$ in S. Notice that, $d(x, m(A)) > 0$, otherwise it implies S is overlapping with $m(A)$, for any

$u \in S$ and $v \in LR_B(u)$, by Lemma 2.5.4, we have $d(v, m(B)) = 0$, but this implies $rev(u, v) = 1$. Therefore, $d(x, m(A)) > 0$.

Lemma 2.5.5 implies that $\forall u \in S$, $d(u, x) \leq \frac{2}{9} d(x, m(A))$. This result tells us the set S is contained in a ball centered at x with radius $\frac{2}{9} d(x, m(A))$. So we can bound the distance between centroid of S, $m(S)$ and $m(A)$ using convexity of the l_2 norm. The following lemma's proof is omitted due to space.

Lemma 2.5.6. Let S be the low-revenue points in A and $x \in \arg\max_{u \in S} d(u, m(A))$. It is the case that $d(m(S), m(A)) \leq \frac{7}{9} d(x, m(A))$.

Proof. This is proved by combining Lemma 2.5.5 with the convexity of l_2 norm. Notice that $m(A)$ is a convex combination of all points in A, Jensen's inequality gives us the conclusion. Since we proved $m(S)$ is far from $m(A)$.

$$d(m(S), x) = d\left(\frac{\sum_{u \in S} u}{|S|}, x\right) \leq \frac{\sum_{u \in S} d(u, x)}{|S|} \leq \frac{2}{9} d(x, m(A))$$

As a result the triangle inequality gives us $d(m(S), m(A)) \geq d(x, m(A)) - d(x, m(S)) \geq \frac{7}{9} d(x, m(A))$. □

Next, we upper-bound $d(m(S), m(B))$. Recall that points in the set S are far away from $m(A)$, but close to the hyperplane $H = \{y : d(y, m(A)) = d(y, m(B))\}$. The following lemma's proof is in the appendix.

Lemma 2.5.7. Let S be the low-revenue points in A. For any $u \in S$, $d(u, m(B)) \leq \frac{11}{9} d(x, m(A))$.

Proof. Since $u \in S$, there exists $w \in B$, s.t. $d(u, w) < \frac{1}{10} \max\{d(u, m(A)), d(w, m(B))\}$. By triangle inequality, we have $d(u, m(B)) \leq d(w, m(B)) + d(u, w)$. Since $rev(u, w) < \frac{1}{10}$, by Lemma 2.5.4, $d(w, m(B)) \leq \frac{10}{9} d(u, m(A))$ and $d(u, w) \leq \frac{1}{9} d(u, m(A))$. Therefore, $d(u, m(B)) \leq \frac{11}{9} d(u, m(A)) \leq \frac{11}{9} d(x, m(A))$. □

Therefore, we can upper bound $d(m(S), m(B))$: $d(m(S), m(B)) \leq \frac{\sum_{u \in S} d(u, m(B))}{|S|} \leq \frac{11}{9} d(x, m(A))$. The first inequality follows by definition of a centroid. The second from Lemma 2.5.7. This, combined with $d(m(S), m(A)) \geq \frac{7}{9} d(x, m(A))$ from Lemma 2.5.6, gives us the following: $\frac{d^2(m(S), m(A))}{d^2(m(S), m(B))} \geq (\frac{7}{9})^2 / (\frac{11}{9})^2 = \frac{49}{121}$. Recall that $D_k(U)$ denotes the optimal k-means value for a set U. Let S_1 and S_2 be two sets. We quote the following lemma from Ostrovsky et al. [119].

Lemma 2.5.8 ([119]). For any two sets of points S_1 and S_2 it is the case that $D_1(S_1 \cup S_2) = D_1(S_1) + D_1(S_2) + \frac{|S_1||S_2|}{|S_1| + |S_2|} d^2(m(S_1), m(S_2))$.

With this in place Lemma 2.5.3 can be shown. In general, we show this by take the set S away from A and assign it into cluster B instead, and prove that this is a better 2-means solution than the previous one.

Proof of [Lemma 2.5.3] Apply Lemma 2.5.8 to $D_1(A) = D_1(S \cup (A \cap S))$, and we get the following equation where obj_1 is the 2-means objective for the solution A, B:

$$obj_1 := D_1(A) + D_1(B) = D_1(S) + D_1(A \cap S) + \frac{|S|(|A| - |S|)}{|A|} d^2(m(S), m(A \cap S)) + D_1(B) \quad (2.1)$$

We want to bound $d^2(m(S), m(A \cap S))$ from the prior equation. By definition of a centroid we know $|S|m(S) + (|A| - |S|)m(A \cap S) = |A|m(A)$. Subtracting $|A|m(S)$ from both sides gives $|A|(m(A) - m(S)) = (|A| - |S|)(m(A \cap S) - m(S))$. Thus we have $d^2(m(S), m(A \cap S)) = \frac{|A|^2}{(|A| - |S|)^2} d^2(m(S), m(A))$. Plugging this into (2.1) and we get

$$\begin{aligned} obj_1 &= D_1(S) + D_1(A \cap S) + D_1(B) + \frac{|S||A|}{|A| - |S|} d^2(m(S), m(A)) \\ &> D_1(S) + D_1(A \cap S) + D_1(B) + \frac{7}{4}|S| d^2(m(S), m(A)) \end{aligned}$$

The last inequality comes from the fact that $|S| > \frac{3}{7}|A|$, so $\frac{|A|}{|A| - |S|} \quad \frac{7}{4}$. Now we consider another way of splitting $A \cup B$! $(A \cap S, S \cup B)$. Note that the following obj_2 is the objective of an alternative 2-means solution: $obj_2 = D_1(A \cap S) + D_1(S \cup B) = D_1(A \cap S) + D_1(S) + D_1(B) + \frac{|S||B|}{|S| + |B|} d^2(m(S), m(B))$. Now we make use of the assumption that $|A| - |B|$. Then $|S| \quad \frac{3}{7}|A| \quad \frac{3}{7}|B|$. Then $\frac{|B|}{|S| + |B|} \quad \frac{1}{3=7+1} = \frac{7}{10}$. So,

$$\begin{aligned} obj_2 &= D_1(A \cap S) + D_1(S) + D_1(B) + \frac{|S||B|}{|S| + |B|} d^2(m(S), m(B)) \\ &\quad D_1(A \cap S) + D_1(S) + D_1(B) + \frac{7}{10}|S| d^2(m(S), m(B)) \\ &\quad D_1(A \cap S) + D_1(S) + D_1(B) + \frac{7}{10}|S| \quad \frac{121}{49} d^2(m(S), m(A)) \\ &< D_1(A \cap S) + D_1(S) + D_1(B) + \frac{7}{4}|S| d^2(m(S), m(A)) = obj_1 \end{aligned}$$

This contradicts the assumption that the split $A \cup B$! (A, B) is the optimal 2-means. Therefore, at least $\frac{4}{7}|A|$ points in A are high-revenue points. □

2.6 Randomly Partitioning Poorly Approximates the Hierarchical-Split Objective

Consider the following algorithm which can create undesirable trees. The Random algorithm splits a set S into (S_1, S_2) by flipping an independent, fair coin for each point in S. If the coin comes up heads then the point gets added to S_1, and otherwise gets added to S_2. The algorithm is intuitively undesirable because it does not take the structure of the input into the construction of the solution. Further, the solutions produced do not give much insight into the data.

While intuitively bad, this algorithm is known to be a $\frac{1}{3}$-approximation for the objective of Moseley and Wang [115] with similarity scores and it is a $\frac{2}{3}$-approximation for value objective for dissimilarity scores. These results hold for any set of similarity or dissimilarity scores, regardless of if they form a metric.

We show that the our objective does not have this shortcoming. The approximation ratio of the Random algorithm is at most $O(\frac{1}{n^\epsilon})$ for a constant $\epsilon > 0$, indicating that it performs very poorly, as is stated by Theorem 2.6.1.

Theorem 2.6.1. Let $OPT(V)$ be the optimal solution for V. Let the expected revenue be $E_T[rev_T(V)]$ for set V. Then, there exists a construction of V, such that for a constant $\epsilon \in (0, 1)$, $E_T[rev_T(V)] = O(\frac{1}{n^\epsilon}) \cdot OPT(V)$.

Before we argue Random is bad, we give the definition of "clean split". Intuitively, a split should be considered clean if it doesn't separate points close to each other when there are far away pairs.

Definition 2.6.1. We define a split $S \to (S_1, S_2)$ to be *clean* if it satisfies one of the following conditions:

1. If $S \subseteq A$ or $S \subseteq B$.
2. If $S_1 \subseteq A, S_2 \subseteq B$, or $S_1 \subseteq B, S_2 \subseteq A$.

Based on the result that every tree is gaining full revenue for an ultrametric, it is easy to see that optimal tree can get a revenue of $OPT(V) := \frac{(n^2+n)(n^2+n-1)}{2} = Q(n^4)$ for the whole point set. The optimal tree splits A from B in the root split, and then can do anything on the remaining portion of the tree.

Before formally prove this theorem we make some quick observations. First, we don't need to care about the pairs (i, j) where $i \in A$ and $j \in B$ because the number of such pairs is $Q(n^3)$, even if we gain full revenue for them, it doesn't affect the approximation ratio.

For the same reason we don't care about pairs (i, j) such that $i, j \in B$. So, we only need to discuss how much revenue we can get from separating all the pairs A inside expectation for Random.

With this in mind, we will use Chernoff bounds to argue that $O(\log n)$ rounds, Random splits each node in half with high probability, which causes us to lose a lot of revenue.

Lemma 2.6.2. Suppose we have a set S with m points, and use Random to split it into S_1 and S_2. Then, for $i = 1, 2$

$$P(|S_i| \geq \frac{m}{2} + \sqrt{m \log m}) \leq \frac{2}{m^2}$$

Proof. Consider m i.i.d. Rademacher variables X_j. Then from Chernoff's bound, we know that

$$P(|\sum_{j=1}^{m} X_j| \geq t) \leq 2\exp(-\frac{t^2}{2m})$$

Random is treating each point j as a Rademacher variable by assigning

$$X_j = \begin{cases} +1 & \text{if } j \text{ is assigned to } S_1 \\ -1 & \text{if } j \text{ is assigned to } S_2 \end{cases}$$

Then, for $i = 1, 2$,

$$P(|S_i| \geq \frac{m}{2} + \sqrt{m \log(m)}) = P(|\sum_{i=1}^{m} X_i| \geq 2\sqrt{m \log(m)})$$
$$\leq 2\exp(-2\log(m)) = \frac{2}{m^2}$$

□

Next, we define "almost-equal" splits, which refers splits such that the points from A and B in the parent node is almost split equally in its two children.

Definition 2.6.2. Given a set S, let S^A and S^B denote the points from A and B in S, respectively. If a split $S \to (S_1, S_2)$ satisfies the property in Lemma 2.6.2, i.e., for $i = 1, 2$, let S_i^A and S_i^B denote the set of points from A and B in set S_i respectively, we say this split is *almost equal* if for $i = 1, 2$:

1. $|S_i^A| \leq \frac{1}{2}|S^A| + \sqrt{|S^A| \log |S^A|}$
2. $|S_i^B| \leq \frac{1}{2}|S^B| + \sqrt{|S^B| \log |S^B|}$

Also, for a hierarchical clustering tree, if all the nodes in the first i layers are almost equally split, we call this tree i-*almost equally split*

The next lemma bounds the number of points in both A and B in an internal node in i^{th} layer if every split is almost equal for both in the first $\frac{\log(n)}{2}$ layers in the tree, where $i \le \frac{\log(n)}{2}$.

Lemma 2.6.3. Let S_i be a node in the i^{th} layer of the tree ($i \le \log n/2$). If all the ancestors of S_i is almost-equally split, let S_i^A be the points in S_i in A, and S_i^B be the points in S_i in B. Then we have $|S_i^A| = Q(n^2/2^i)$, $|S_i^B| = Q(n/2^i)$.

Proof. By induction, we prove a stronger conclusion:

$$\frac{n^2}{2^i} - 8\sqrt{\frac{n^2}{2^i}\log(\frac{n^2}{2^i})} \le |S_i^A| \le \frac{n^2}{2^i} + 8\sqrt{\frac{n^2}{2^i}\log(\frac{n^2}{2^i})}$$

and

$$\frac{n}{2^i} - 8\sqrt{\frac{n}{2^i}\log(\frac{n}{2^i})} \le |S_i^B| \le \frac{n}{2^i} + 8\sqrt{\frac{n}{2^i}\log(\frac{n}{2^i})}$$

We just prove the first claim and the other can be proved in the same way. By induction,

$$\begin{aligned}
|S_i^A| &\le \frac{|S_{i-1}^A|}{2} + \sqrt{|S_{i-1}^A|\log(|S_{i-1}^A|)} \\
&\le \frac{1}{2}\left(\frac{n^2}{2^{i-1}} + 8\sqrt{\frac{n^2}{2^{i-1}}\log(\frac{n^2}{2^{i-1}})}\right) \\
&\quad + \sqrt{\frac{n^2}{2^{i-1}}\log(\frac{n^2}{2^{i-1}})} \\
&= \frac{n^2}{2^i} + 5\sqrt{2}\sqrt{\frac{n^2}{2^i}(\log(\frac{n^2}{2^i}) + \log(2))} \\
&\le \frac{n^2}{2^i} + 8\sqrt{\frac{n^2}{2^i}\log(\frac{n^2}{2^i})}
\end{aligned}$$

And the other side of the inequality can be bounded in the same way. □

If the condition in Lemma 2.6.3 holds, this result tells us that every node in the first $\frac{\log(n)}{2}$ layers is not clean. In other words, for all the pairs of points in A which are separated during the first $\frac{\log(n)}{2}$ layers of the tree, we don't get any revenue. Thus we can upper bound the revenue for points in A:

Lemma 2.6.4. If the tree T is $\frac{\log(n)}{2}$-almost-equally-split tree, for all the pairs in A the revenue is $Q(n^{4-e})$ for $e = \frac{\log(2)}{2}$.

Proof. The tree does not obtain any revenue in the splits in the first $(\frac{\log(n)}{2})^{th}$ layers. It can only get revenue within the subtrees rooted at each internal node in the $(\frac{\log(n)}{2})^{th}$ layer. For each such node, the size of the subtree is $Q(\frac{n^2}{2^{\log(n)/2}}) = Q(n^{2-e})$, where $e = \frac{\log 2}{2}$. Thus the

revenue we can get from each subtree is bounded by $O(n^{4-2\epsilon})$. There are $O(n^{\epsilon})$ such nodes. So, the revenue is bounded by $O(n^{4-\epsilon})$. □

We have already proved that if many of the top layers have almost equally split internal nodes, the hierarchical clustering tree has small total revenue. To formally prove Theorem 2.6.1, we only need to show that this happens with high probability. Notice that the probability of the tree being not $\frac{\log(n)}{2}$-almost equally split can be bounded by union bounds on the probability of an almost equal split not happening in any of the first $\frac{\log(n)}{2}$ layers, which is $O(\frac{1}{n^{\epsilon'}})$, where $\epsilon' = 2 - \frac{3\log 2}{2}$. This is very low probability, putting everything together, we have Lemma 2.6.4.

Proof of [Theorem 2.6.1] By Lemma 2.6.4,

$$E_T(rev_T(V) | T \text{ is } \frac{\log(n)}{2}\text{-almost equally split}) = O(n^{4-\epsilon})$$

Then, we only need to lower bound the probability that the tree is $\frac{\log(m)}{2}$-almost equally split. We show next that this happens with very high probability. Again, let S_i denote some node in the i^{th} layer of T.

$$P(S_i \text{ isn't almost equal split} | T \text{ is } (i-1)\text{-almost equally split}) \leq \frac{2}{|S_i^B|} + \frac{2}{|S_i^A|} \leq Q(\frac{2^i}{n})$$

In the i^{th} layer, we have 2^i nodes. So we bound the probability of having a tree that's almost equal split in the first $\frac{\log(n)}{2}$ layers as follows:

$$
\begin{aligned}
& P(T \text{ is } i\text{-almost equally split}) \\
= & \prod_{i=1}^{\frac{\log(n)}{2}} P_{S_i \text{ in the } i^{th} \text{ layer}} P(S_i \text{ is almost equally split} \mid T \text{ is } (i-1)\text{-almost equally split}) \\
> & \prod_{i=1}^{\frac{\log(n)}{2}} P_{S_i \text{ in the } i^{th} \text{ layer}} (1 - Q(\frac{2^i}{n})) \\
> & 1 - Q(\sum_{i=1}^{\frac{\log(n)}{2}} \sum_{S_i \text{ in the } i^{th} \text{ layer}} (\frac{2^i}{n})^2) \\
= & 1 - Q(\sum_{i=1}^{\frac{\log(n)}{2}} 2^i \frac{4^i}{n^2}) \\
= & 1 - Q(\frac{(1 + 8 + 8^2 + \cdots + 8^{\frac{\log(n)}{2}})}{n^2}) \\
= & 1 - Q(2^{\frac{3\log(n)}{2}} = n^2) = 1 - Q(\frac{1}{n^{e'}})
\end{aligned}
$$

Where $e' = 2 - \frac{3\log 2}{2} > e$. So we have $Q(\frac{1}{n^{e'}})$ probability that T is not i-almost equally split, in which case the revenue is bounded by $Q(n^4)$.

Therefore, the expectation is bounded by:

$$
\begin{aligned}
& E_T[rev_T(V)] \\
& P(T \text{ is } \frac{\log(n)}{2}\text{-almost equally split}) Q(n^{4-e}) \\
& + P(T \text{ is not } \frac{\log(n)}{2}\text{-almost equally split}) Q(n^4) \\
& Q(n^{4-e}) + Q(n^{4-e'}) \\
& Q(n^{4-e})
\end{aligned}
$$

where $e = \frac{\log(2)}{2}$. □

2.7 Other Objectives for Data in Metric Space

This section studies data with similarity/dissimilarity scores in a metric space. First we investigate the value objective from [58]. Recall that this is the same as cost from [61] except

the minimization is swapped for a maximization and the similarity scores are swapped for dissimilarity scores. We also study cost objective for similarity scores. We show for each case that if the pairwise similarity/dissimilarity scores form a metric, then every tree is a at most a factor 2 from optimal.

For a tree T let $T[i_j]$ denote the subtree rooted at the least common ancestor of i and j, and $|leaves(T[i_j])|$ denote the number of leaves of $T[i_j]$. Recall that the value objective is the following: $\max \sum_{i,j \in V} d(i,j)|leaves(T[i_j])|$.

In Cohen-Addad et al. [58], it has been proved that any solution is a constant approximation of the optimal solution for the value objective, given that the distance is a metric. Here we prove a stronger conclusion:

Theorem 2.7.1. *Any solution is a $\frac{1}{2}$-approximation for value objective if the distance $d(i,j)$ satisfies triangle inequality.*

To prove this theorem, fix a tree T. Let $LCA(i,j)$ be the least common ancestor of i and j in T. Let $\mathbb{1}\{i,j|k\}$ be an indicator variable indicating whether in the tree T, the $LCA(i,j)$ is a descendant of $LCA(i,j,k)$: $\mathbb{1}\{i,j|k\} = 1$ if such a relationship holds, and $\mathbb{1}\{i,j|k\} = 0$ if otherwise. Equivalently, if $\mathbb{1}\{i,j|k\} = 1$, it means during the tree construction, tracing down from the root, k is separated from i and j first, and i,j are separated in a split closer to the bottom of the tree. If the tree is binary, we have the following equality:

$$\mathbb{1}\{i,j|k\} + \mathbb{1}\{i,k|j\} + \mathbb{1}\{j,k|i\} = 1$$

That is, one and only one of the relationships represented by the three indicator variables can hold.

Prior to proving Theorem 2.7.1, we cite this result from Wang and Wang [131], which shows the revenue in Cohen-Addad et al. objective can be decomposed onto every triangle:

Lemma 2.7.2([131]). *When $|V| \geq 3$,*

$$\begin{aligned} R_T(V) &= \sum_{i,j \in V} d(i,j)|leaves\, T[i_j]| \\ &= \sum_{\{i,j,k\} \subseteq V} tri_{R_T}(i,j,k) + 2 \sum_{\{p,q\} \subseteq V} d(p,q) \end{aligned}$$

where $triR_T(i,j,k)$ denotes the revenue on triangle i,j,k, de ned as follows:

$$triR_T(i,j,k) = \begin{cases} d(i,k) + d(j,k) & \text{if } 1\{i,j \| k\} = 1 \\ d(i,j) + d(j,k) & \text{if } 1\{i,k \| j\} = 1 \\ d(i,j) + d(i,k) & \text{if } 1\{j,k \| i\} = 1 \end{cases}$$

By triangle inequality, for each triangle i,j,k, we always have $triR_T(i,j,k) \geq \frac{1}{2}(d(i,k) + d(j,k) + d(i,j))$, which will give us Theorem 2.7.1.

Proof of [Theorem 2.7.1] Let $OPT(V)$ denote the optimal value of Cohen-Addad et al. objective for V. We have $OPT(V) = \sum_{\{p,q\} \subseteq V} |V| \, d(p,q)$.

By triangle inequality, it is easy to see that regardless of which of the three relationship holds, we always have $triR_T(i,j,k) \geq \frac{1}{2}(d(i,j) + d(i,k) + d(j,k))$ for any triplet $\{i,j,k\}$. Then, for any T,

$$\begin{aligned} R_T(V) &= \sum_{\{i,j,k\} \subseteq V} triR_T(i,j,k) + 2 \sum_{\{p,q\} \subseteq V} d(p,q) \\ &\leq \sum_{\{i,j,k\} \subseteq V} \frac{1}{2}(d(i,j) + d(i,k) + d(j,k)) \\ &\quad + 2 \sum_{\{p,q\} \subseteq V} d(p,q) \\ &= \frac{1}{2} \sum_{\{p,q\} \subseteq V} d(p,q)(|V| - 2) + 2 \sum_{\{p,q\} \subseteq V} d(p,q) \\ &< \frac{1}{2} \sum_{\{p,q\} \subseteq V} |V| \, d(p,q) = \frac{1}{2} OPT(V) \end{aligned}$$

□

Next consider the objective in Dasgupta [61]. Here each pair of points i and j have a similarity score w_{ij} where higher weights mean points are more similar. Recall from the introduction that Dasgupta's objective is $\min_T \sum_{i,j \in V} s(i,j) |leaves(T[i \vee j])|$.

The following corollary easily follows from the proof to Theorem 2.7.1.

Corollary 2.7.2.1. If the similarity score in the setting of Dasgupta [61] is a metric, any hierarchical clustering tree is a 2-approximation for the objective in Dasgupta [61]: $\min_T cost_T(V) = \sum_{1 \leq i < j \leq n} s(i,j) |leaves(T[i \vee j])|$.

We note that for similarity scores, it is not a standard assumption that data lies in a metric space. Thus, this corollary is perhaps interesting to understand the structure of the objective.

However, it does not suggest that any tree will be a 2-approximate for most data sets with similarity scores.

2.8 Empirical Results

The goal of this section is to study the performance of different algorithms for the new objective empirically. The experimental results support the following claims:

- Algorithms that are popular in practice give high revenue for the new objective, with bisecting k-means performing the best. This demonstrates that the objective value is correlated with algorithms that perform well and highly connected to the bisecting k-means algorithm, as the theory suggests.
- Random algorithm, as mentioned in previous section, performs poorly for the new objective.

Data sets. We use two data sets from the UCI data repository[7]: Census and Bank[8]. Only the numerical features are used.

Algorithm	$(\hat{m}_1, \hat{s}_1)$-Census	$(\hat{m}_2, \hat{s}_2)$-Census	$(\hat{m}_1, \hat{s}_1)$-Bank	$(\hat{m}_2, \hat{s}_2)$-Bank
bisecting k-means	(4.931e5, 304.980)	(1.094e12, 1.714e11)	(4.912e5, 474.451)	(1.049e12, 1.158e11)
average-linkage	(4.900e5, 1.151e3)	(1.093e12, 1.710e11)	(4.907e5, 802.665)	(1.052e12, 1.163e11)
single-linkage	(4.869e5, 1.392e3)	(1.094e12, 1.712e11)	(4.818e5, 1.365e3)	(1.035e12, 1.168e11)
Random	(1.311e5, 1.072e4)	(7.463e11, 1.152e11)	(3.339e5, 8.825e3)	(7.789e11, 7.993e10)
upper bound	(499500, 0)	(1.119e12, 1.725e11)	(499500, 0)	(1.167e12, 1.199e11)

Table 2.1 Summary of stats for all algorithms, on Census and Bank

Setting. We study four algorithms[9]: bisecting k-means, average-linkage, single-linkage, and Random. In each experiment, we subsample 2000 data points from the given data set and run the algorithms with subsampled data. We conduct five experiments with each data set and report the mean and variance. Since optimal 2-means solution is intractible, in practice we import the k-means implementation from package Scikit-learn[10], which uses Lloyd's algorithm seeded with k-means++ for each split.

[7] https://archive.ics.uci.edu/ml/datasets/census+income
[8] https://archive.ics.uci.edu/ml/datasets/Bank+Marketing
[9] https://github.com/wang-yuyan/hier_clustering_split_rev_obj_test.git
[10] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

Results. Table 2.1 shows the comparison between performance for our objective and the value objective. The rows denote the results for different algorithms and the upper-bound for both value and our objective. For each algorithm, the columns $(\hat{m}_1, \hat{s}_1)$ and $(\hat{m}_2, \hat{s}_2)$ denote the mean and standard deviation for our objective and value objective respectively, calculated over results of the five experiments.

Regarding the new objective, the results show bisecting k-means performs the best of the four algorithms for it. Further, bisecting k-means is within 1% of the upper bound on the optimal solution for the objective, which is the total number of pairs of data points. This suggests that the objective is closely related to bisecting k-means, as the theory suggests. It also shows that experimentally bisecting k-means performs much better than the approximation ratio established.

All the three algorithms which are popular in practice perform well for our objective, with bisecting k-means performing marginally better than average-linkage and single-linkage on average. Moreover, bisecting k-means also has the smallest standard deviation across different subsamples. Random is significantly worse, with potentially over 30 times more loss compared to optimal than the other algorithms. This perhaps suggests that trees created by good algorithms perform well for the objective and poorly constructed trees do not perform well.

Compared with the value objective from prior work, the results further show that average-linkage performs slightly better than bisecting k-means for the value objective. This result matches the theory, which suggests this objective is closer to average-linkage than bisecting k-means. Again, all three algorithms used in practice perform well for value. However, Random also gives about $2\times$ of the upper-bound, as the theoretical bound suggests. This perhaps shows the value objective gives similar judgements on algorithm performance with our objective, the latter showing a more significant gap between Random and the other three algorithms.

2.9 Conclusion

This chapter gives a new objective function for hierarchical clustering designed to mathematically capture the principle used to motivate most divisive algorithms. That is, comparing inter vs. intra cluster distances at splits in the tree.

The paper proved a close relationship between the objective and the bisecting k-means algorithm. This was done by showing the bisecting k-means provably optimizes the objective. This helps to understand the structures of trees produced using bisecting k-means.

The results in this paper leave directions for future work. How tight can the approximation ratio be for the k-means algorithm? How do other hierarchical clustering algorithms perform for this objective? Can we improve on the bisecting k-means algorithm to better optimize the objective?

Chapter 3

Fair Hierarchical Clustering

3.1 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 [87]. 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. [30]).

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 [15, 16, 25, 32, 33, 48, 55, 76, 82, 88, 89]. However, the same issues affect hierarchical clustering, which is the problem we study in our paper [14], which this chapter is based on.

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 bene t from fair hierarchical clustering, motivating its study.

Our contributions. We initiate an algorithmic study of fair hierarchical clustering. Fairness is imposed as a constraint on the global hierarchical clustering tree while the tree's clustering performance is measured using the revenue [115], value [58], and cost [61] objectives as stated in 2.2.

To achieve fairness, we show how to extend the fairlets machinery, introduced by Chierichetti et al. [52] and extended by Ahmadian et al. [15], to this problem. We then investigate the complexity of nding a good fairlet decomposition, giving both strong computational lower bounds and polynomial time approximation algorithms. Based on the fairlet decomposition, we show how to build fair hierarchical clustering trees with theoretical performance guarantees for the objectives.

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 negligible objective degradation.

Related work. Hierarchical clustering has received increased attention over the past few years. In Section 2.2 we have seen recent developments in hierarchical clustering, including new objective function designs and analysis of algorithms utilizing the new framework.

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 [30] for a rich overview, and focus here on progress for fair clustering. Chierichetti et al. [52] rst de ned fairness for k-median and k-center clustering, and introduced the notion of fairlets to design ef cient algorithms. Extensive research has focused on two topics: adapting the de nition of fairness to broader contexts, and designing ef cient algorithms for nding good fairlet decompositions. For the rst topic, the fairness de nition was extended to multiple values for the protected feature, e.g. [115, 13, 122]. For the second topic, Backurs et al. [25] proposed a near-linear constant approximation algorithm for nding fairlets for k-median, Schmidt et al. [127] introduced a streaming algorithm for scalable computation of coresets for fair clustering, Kleindessner et al. [88] designed a linear time constant approximation algorithm for k-center where cluster centers are selected proportionally from a set of colors, Bercea et al. [33] developed methods for fair k-means, while Ahmadian et al. [16] and Ahmadi et al. [12] de ned approximation algorithms for fair correlation clustering. Concurrently with our work, Chhabra et al. [51] introduced a possible

approach to ensuring fairness in hierarchical clustering. However, their fairness de nition 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 [40], fair optimization [53], as well as other problems [41].

3.2 Formulation

3.2.1 Generalized Objectives for Hierarchical Clustering

In this section we reiterate the objective functions proposed in 2.2, but generalize them to weighted inputs. This chapter will need to use the generalized form of these objectives. As introduced in Section 2.2, we let $G = (V; V^2 \to \mathbb{R}^{\geq 0})$ denote an input instance, where V is a set of data points equipped with similarity/dissimilarity scores $V^2 \to \mathbb{R}^{\geq 0}$. We denote by $n = |V|$, the number of points. We then consider the vertex-weighted versions of the problem. Here, we have $G = (V; V^2 \to \mathbb{R}^{\geq 0}; m)$, 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 $m(U) = \sum_{i \in U} m(i)$.

We again use $s(\cdot, \cdot)$ to denote similarity scores and $d(\cdot, \cdot)$ to denote dissimilarity/distance scores.

We consider the three different objectives —*revenue*, *value*, and *cost*—, and generalize them to the vertex-weighted case.

De nition 3.2.1 (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 $\text{rev}_G(T) = \sum_{i,j \in V} s(i, j) \cdot m(V) - m(\text{leaves}(T[i \vee j]))$:

Note that in this de nition, each weight is scaled by (the vertex-weight of) all the non-leaves. The goal is to nd a tree of maximum revenue. Moseley and Wang [115] showed that average-linkage is a $\frac{1}{3}$-approximation for vertex-unweighted revenue; the state-of-the-art is a $(\frac{1}{3} + \varepsilon)$-approximation by Charikar et al. [44]. As part of the analysis, there is an upper bound for the revenue objective Cohen-Addad et al. [58], Moseley and Wang [115], which is easily extended to the vertex-weighted setting:

$$\text{rev}_G(T) \leq m(V) \min_{u,v \in V; u \neq v} m(\{u, v\}) \cdot s(V). \tag{3.1}$$

Note that in the vertex-unweighted case, the upper bound is $\frac{1}{2}|V|(|V| - 2)s(V)$.

De nition 3.2.2 (Value). The value (val) of a tree T for an instance $G = (V; d; m)$ where $d(\ ;\ )$ denotes distance is:

$$val_G(T) = \sum_{i;j \in V} d(i;j) \cdot m(leaves(T[i \vee j])):$$

As in revenue, we aim to nd a hierarchical clustering to maximize value. Cohen-Addad et al. [58] showed that both average-linkage and a locally densest cut algorithm achieve a $2=3$-approximation for vertex-unweighted value. They also provided an upper bound for value, much like that in (3.1), which in the vertex-weighted context, is:

$$val_G(T) \quad m(V) \cdot s(V): \tag{3.2}$$

De nition 3.2.3 (Cost). The cost of a tree T for an instance $G = (V; s)$ where $s(\ ;\ )$ denotes similarity is:

$$cost_G(T) = \sum_{i;j \in V} s(i;j) \cdot |leaves(T[i \vee j])|:$$

Here we note that from a complexity point of view, cost is a much more dif cult objective to optimize. Charikar and Chatziafratis [43] 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.

3.2.2 Notions of Fairness

Many de nitions 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 value of 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. Chierichetti et al. [52] formalized this notion 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 de ned by Ahmadian et al. [15]. 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. 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 nice property:

Definition 3.2.4 (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 particularly useful in hierarchical clustering: given a tree and internal node u which represents a cluster, if each child cluster of u is fair, then u must also be a fair cluster. We can now extend the notion of fair clustering to hierarchical clustering.

Definition 3.2.5 (Fair hierarchical clustering)**.** For any fairness constraint, a hierarchical clustering is fair if all of its clusters (besides the leaves) are fair, and the minimal non-leaf clusters form a clustering of all the data.

Thus, under any union-closed fairness constraint, this definition is equivalent to restricting the bottom-most clustering (besides the leaves) to be fair. Given an objective (like revenue or value), the goal then is to find a fair hierarchical clustering that optimizes the objective. For ease of exposition, we focus on the bounded representation fairness notion with $\beta = 1$ and an over-representation cap α. However, the main ingredient for the revenue and value objectives works under any notion of fairness that is union-closed. In the following sections, we will focus on optimtizing value and revenue objectives with the fairness constraint. We save the arguments for the more challenging cost objective until the full paper due to space limit.

3.3 Fairlet Decomposition

We recall the notion of fairlets in Chierichetti et al. [52].

Definition 3.3.1 (Fairlet)**.** 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 other words, a fairlet is a minimal fair set that cannot be further decomposed.

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 following lemma shows the fairlet size will always be at most $2t-1$. We will refer to the maximum possible size of a fairlet by $(m_f = 2t-1)$.

Lemma 3.3.1. 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 $t \le |P_i| < 2t$.

Proof. Let $p = mt + r$ with $0 \le r < t$, 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 $1 \le \frac{1}{t}|P_i|$. □

Recall that given a union-closed fairness constraint, a hierarchical clustering tree is fair if and only if the first clustering in the tree is a layer of fairlets, which we call a fairlet decomposition, of the original data set. 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 3.4, 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 as the performance of the tree built on top of these fairlets are more closely tied to the quality of the decomposition. As we shall see in Section 3.3.1, we set a separate optimization goal for the desired fairlet decomposition for value.

This chapter will focus on revenue and value. For the cost objective, generally considered the most challenging to optimize or approximate, more assumptions on the input and complex algorithm designs are needed to achieve performance guarantees. The results for cost could be found in the full version of [14].

3.3.1 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 $Y = \{Y_1, Y_2, \ldots\}$ such that Y_i is an α-capped fairlet. The cost of this decomposition is defined as:

$$f(Y) = \sum_{Y \in Y} \sum_{\{u,v\} \subseteq Y} d(u,v). \tag{3.3}$$

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

Theorem 3.3.2. Let $z \geq 3$ be an integer. Then there is no bounded approximation algorithm for finding $(\frac{z}{z+1})$-capped fairlets optimizing $f(Y)$, which runs in polynomial time, unless P=NP.

The proof proceeds by a reduction from the Triangle Partition problem, which asks whether a graph $G = (V, E)$ on $3n$ vertices can be partitioned into three element sets, with each set forming a triangle in G.

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

Definition 3.3.2. 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 3.3.3. 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 3.3.3. For a fixed constant k, 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 $V' = V \cup C_1 \cup C_2 \cup \ldots \cup C_n$, 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 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 $C_i \subset S_i$, then $V = \{S'_1 \cap C_1, \ldots, S'_n \cap C_n\}$ is a triangle partition. □

We are now ready to prove Theorem 3.3.2.

Proof of [Theorem 3.3.2] We prove Theorem 3.3.2 by proving that for given (r, n) where $r \geq 3$, if there exists a c-approximation polynomial algorithm $\mathcal{A}$ for (3.3), it can be used to solve the PIKC problem where $k = r$ for any instance as well. This holds for any n and c.

Given any graph $G = (V, E)$ that is input to the PIKC problem, where $|V| = kn = rn$, 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 set of b points.
2. Color the points in V red, and color all the C_i's blue.
3. For $e = (u, v)$, let $w(u, v) = 0$, 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 balance of V' is $\frac{b}{r}$, so each fairlet should have exactly b blue points and r red points.

We claim that G has a k-clique partition if and only if algorithm $\mathcal{A}$ gives a solution of 0 for (3.3). The same argument as in the proof of Lemma 3.3.3 will show that G has a k-clique partition if and only if the optimal solution to (3.3) is 0. This is equal to algorithm $\mathcal{A}$ giving a solution of 0 since otherwise the approximate is not bounded. □

Fortunately, for the purpose of optimizing the value objective, it is not necessary to find an approximate decomposition.

3.4 Optimizing revenue with fairness

In this section we consider 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 α-approximation algorithm to (3.1) plus this fairlet decomposition can be used to obtain a (roughly) α-approximation for fair hierarchical clustering under the revenue objective, and (iii) use average-linkage, which is known to be a $1/3$-approximation to (3.1).

First, we address step (ii).

Theorem 3.4.1. Given an algorithm that obtains a α-approximation to (3.1) where $\alpha \geq 1$, and a fairlet decomposition with maximum fairlet size m_f, there is a $\left(1 - \frac{2m_f}{n}\right)\alpha$-approximation for fair hierarchical clustering under the revenue objective.

Proof. Let A be the b-approximation algorithm to (3.1). 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 input instance $G_Y = (\mathcal{Y}, s_Y, m_Y)$. For $Y, Y' \in \mathcal{Y}$, we define $s(Y, Y') = \sum_{i \in Y, j \in Y'} s(i, j)$ and we define $m_Y(Y) = |Y|$.

We run A on G_Y and let T_Y be the hierarchical clustering obtained by 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_Y.

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

Since A obtains a b-approximation to hierarchical clustering on G_Y, we have

$$\mathrm{rev}_{G_Y}(T_Y) \geq b \sum_{Y, Y' \in \mathcal{Y}} s(Y, Y')(n - m(Y) - m(Y')).$$

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,

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

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

Prior work showed that average-linkage is a $\frac{1}{3}$-approximation to (3.1) in the vertex-unweighted case; this proof can be easily modified to show that it is still a $\frac{1}{3}$-approximation even with vertex weights.

Combined with the fairlet decomposition methods for the two-color case [52] and for multi-color case shown in Lemma 3.3.1, we have the following.

Corollary 3.4.2. 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.

3.5 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 (3.3), and then use any approximation algorithm for weighted hierarchical clustering with the decomposition as the input. Again, our result applies to Bera et al. [32], Bercea et al. [33]'s fairness constraint if we are given an appropriate fairness decomposition.

Theorem 3.5.1. Given an algorithm that gives a b-approximation to (3.2) where $b \geq 1$, and a fairlet decomposition $\mathcal{Y}$ such that $f(\mathcal{Y}) \leq \epsilon\, d(V)$, there is a fair solution which is a $b(1+\epsilon)$ approximation for (3.2).

To prove this theorem, 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 $b\, m(V)\, d(V)$ for the value objective on a weighted input. Recall that we extended the value objective to a weighted input and we denote $m(V) = \sum_{u \in V} m_u$. 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 $b(1-\epsilon)$-approximation for the value objective, if $f(\mathcal{Y}) \leq \epsilon d(V)$.

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 3.5.1. 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.

Claim 3.5.2. On the instance $V(\mathcal{Y})$ the algorithm A has a total weighted objective of $b(1-\epsilon)\, n\, d(V)$.

Proof. Notice that $m(V(\mathcal{Y})) = |V| = n$. Consider the total sum of all the distances in $V(\mathcal{Y})$. This is $\sum_{a_i, a_j \in V(\mathcal{Y})} d(a_i, a_j) = \sum_{Y_i, Y_j \in \mathcal{Y}} d(Y_i, Y_j) = d(V) - f(\mathcal{Y})$. The upper bound on the optimal solution is $(\sum_{Y_i \in \mathcal{Y}} m_i)(d(V) - f(\mathcal{Y})) = n(d(V) - f(\mathcal{Y}))$. Since $f(\mathcal{Y}) \leq \epsilon d(V)$, this upper bound is at least $(1-\epsilon)\, n\, d(V)$. Claim 3.5.2 follows from the fact that the algorithm A archives a weighted revenue at least b factor of the total weighted distances. □

Theorem 3.5.1 is essentially a corollary of Claim 3.5.2. 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 (i.e., different values of the protected feature) and let $Y = \{Y_1, Y_2, \ldots\}$ 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 3.5.3. There exists a local search algorithm that finds a fairlet decomposition Y with $f(Y) \leq (1+\varepsilon)\max_{i,k} \frac{r_{i,k}}{n_i} d(V)$ in time $\tilde{O}(n^3/\varepsilon)$.

We can now use the fact that both average-linkage and the $\frac{\theta}{n}$-locally-densest cut algorithm give a $\frac{2}{3}$- and $(\frac{2}{3} - \varepsilon)$-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 grows, the ratio $r_{i,k}/n_i$ goes to 0. These results, combined with Theorem 3.5.1 and Theorem 3.5.3, yield Corollary 3.5.4.

Corollary 3.5.4. 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 $\frac{\theta}{n}$-locally-densest cut algorithm in Cohen-Addad et al. [58], we get a polynomial time algorithm for fair hierarchical clustering that is a $(\frac{2}{3} - \varepsilon)(1 - o(1))$-approximation under the value objective for any $\varepsilon > 0$.

Given at most a small fraction of every color is in any cluster, Corollary 3.5.4 states that we can extend the state-of-the-art results for value to the 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 3.5.3. This is inspired by the $\frac{\theta}{n}$-densest cut algorithm of Cohen-Addad et al. [58]. 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. [52]. For multiple colors with $\alpha = 1/t$, we use Lemma 3.3.1. Our algorithm will then recursively

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

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

1: Find $d_{\max}$, $D \leftarrow \frac{m_f}{n} d_{\max}$.
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.
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, $\frac{\sum_{Y_k \in \mathcal{Y}'} d(Y_k)}{\sum_{Y_k \in \mathcal{Y}} d(Y_k)} < (1 + \varepsilon/n)$ and $\sum_{Y_k \in \mathcal{Y}} d(Y_k) > D$ **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**

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 3.5.2 (Local optimality)**.** Consider any fairlet decomposition $\mathcal{Y} = \{Y_1, Y_2, \ldots\}$ and $\varepsilon > 0$. 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 $(1 + \varepsilon)$ factor.

The algorithm constructs an (ε/n)-locally optimal algorithm for fairlet decomposition, which runs in $\tilde{O}(n^3/\varepsilon)$ time. Consider any given instance (V, d). Let $d_{\max}$ denote the maximum distance, m_f denote the maximum fairlet size, and $D = d_{\max} \frac{m_f}{n}$. 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 3.5.5. Algorithm 1 finds a valid fairlet decomposition.

We prove two things: Algorithm 1 optimizes the objective (3.3), and has a small running time. The following lemma gives an upper bound on $\mathcal{Y}$'s performance for (3.3) found by Algorithm 1.

Lemma 3.5.6. The fairlet decomposition $\mathcal{Y}$ computed by Algorithm 1 has an objective value for (3.3) of at most $(1 + \varepsilon) \max_{i,k} \frac{r_{i,k}}{n_i} d(V)$.

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

Let $Y : V \to Y$ 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 Y is (ε/n)-locally-optimal; 2) if $\sum_{Y_k \in Y} d(Y_k) \le D$. 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:

$$d(u, Y(u)) + d(v, Y(v)) \le d(u, Y(v)) + d(v, Y(u)) + \frac{\varepsilon}{n} \sum_{Y_k \in Y} d(Y_k). \tag{3.4}$$

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).

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

Divide both sides by n_i and we get:

$$\sum_{Y_k} d_{R_i}(Y_k) \le \left(\sum_{Y_k} \frac{r_{ik}}{n_i} d_{R_i}(Y_k) \right) + \left(\sum_{u \in R_i(V)} \sum_{Y_k \neq Y(u)} \frac{r_{ik}}{n_i} d(u, Y_k) \right) + \frac{n_i \varepsilon}{n} \sum_{Y_k} d(Y_k). \tag{3.5}$$

Now we sum up this inequality over all colors R_is. 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 $\frac{\varepsilon(\sum_i n_i)}{n} \sum_{Y_k} d(Y_k) = \varepsilon \sum_{Y_k} d(Y_k)$. Using the fact that $\frac{r_{ik}}{n_i} \le \max_{i,k} \frac{r_{ik}}{n_i}$, the other terms sum up to :

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

Therefore, putting LHS and RHS together, we get

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

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

□

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 $\frac{2(b+r)}{n}$. If there exists a caplet decomposition such that $\max_{i,k}\frac{r_{ik}}{n_i} = o(1)$, Lemma 3.5.6 implies we can build a fair hierarchical clustering tree with $o(1)$ loss in approximation ratio for value objective.

Assuming for all color classes R_i, $n_i ! +\infty$ as $n ! +\infty$, 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.

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 3.6.

Lemma 3.5.7. The running time for Algorithm 1 is $\tilde{O}(n^3/\epsilon)$.

Proof. 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 $Y_1,\ldots,Y_{t\ 1}$ before the last swap. Since the While loop does not terminate here $\sum_{Y_k} d(Y_k)$ $D = \frac{b+r}{n}d_{max}$. However, at the beginning, we have $\sum_{Y_k} d(Y_k)$ $(b+r)n\ d_{max} = n^2 D$ $n^2\sum_{Y_k} d(Y_k)$. 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. □

Together, Lemma 3.5.6, Lemma 3.5.7, and Claim 3.5.5 prove Theorem 3.5.3. This establishes that there is a local search algorithm that can construct a good fairlet decomposition.

Table 3.1 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 3.2 Impact of Algorithm 1 on ratio r_{value} in percentage (mean 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

3.6 Experiments

This section validates our algorithms from Sections 3.4 and 3.5 empirically. 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 this section we will first study the results for the two-color case in detail, and then show that in general the results extend to multi-color cases.

Datasets. We use two datasets from the UCI data repository[1]. For 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 $s(i, j) = \frac{1}{1+d(i,j)}$ 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 3.1 for details).

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

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

Table 3.3 Impact of Algorithm 1 on 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
nal	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
nal	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
nal	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
nal	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)

Fig. 3.1 (i) ratio$_{\text{fairlets}}$, every 100 swaps. (ii) ratio$_{\text{value}}$, every 100 swaps. (iii) CENSUSGENDER: running time vs sample size on a log-log scale.

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 $2^i; i = 0, \ldots, 8$. On each, we do 5 experiments and report the average results. We use Algorithm 1 to 0.1 in all of the experiments.

Implementation. 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 2 failed attempts to swap a random pair. As we observe empirically, this does not have material effect on the quality of the overall solution.

Metrics. 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 ratio$_{\text{value}} = \frac{\text{value}_G(T)}{\text{value}_G(T^0)}$ for value objective. This measures how close the performance of the found fair tree is to the unfair tree constructed using traditional

Table 3.4 Clustering on fairlets found by local search vs. upper bound, performance ratio in percentage, at size 1600 (mean std. dev).

Dataset	CENSUS GENDER		CENSUS RACE		BANK MARRIAGE		BANK AGE	
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

algorithms. Similarly, for revenue, we consider the ratio $\text{ratio}_{\text{rev}} = \frac{\text{rev}_G(T)}{\text{rev}_G(T^*)}$. We consider the fairlet objective function where Y is a fairlet decomposition. Let $\text{ratio}_{\text{fairlets}} = \frac{f(Y)}{d(V)}$.

Results. The experimental results are shown as follows. We first present results for value objective with two colors, and then extend the results to the revenue objective, and multiple colors. In all settings the results are qualitatively similar.

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 3.2, 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 search algorithm performs swaps. Fairness leads to almost no degradation in the objective value as the swaps increase. Table 3.3 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 (3.3), the better the tree's performance based on the fairlets. Figures 3.1(i) and 3.1(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 2200 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 3.4. On all datasets, we obtain ratios significantly better than the theoretical worst case guarantee. In Figure 3.1(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.

We then switch from value to revenue objective. 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

Table 3.5 Impact of different fairlet decomposition on $\text{ratio}_{\text{rev}}$ in percentage (mean 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
nal	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
nal	85.38	1.68	92.98	1.89	94.99	0.52	96.86	0.85	97.24	0.63
BANK MARRIAGE, initial	75.19	2.53	73.58	1.05	74.03	1.33	73.68	0.59	72.94	0.63
nal	93.88	2.16	96.91	0.99	96.82	0.36	97.05	0.71	97.81	0.49
BANK AGE, initial	77.48	1.45	78.28	1.75	76.40	1.65	75.95	0.77	75.33	0.28
nal	91.26	2.66	95.74	2.17	96.45	1.56	97.31	1.94	97.84	0.92

nal fairlet decomposition found by Algorithm 1 for revenue objective using the converted similarity scores. Table 3.5 shows the ratio of fair tree built by using average-linkage on different fairlet decompositions.

Table 3.6 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;+¥) : 6822	(0;26] : 6413
BANK MULTICOLOR	(30;38] : 14845	(38;48] : 12148	(48;+¥) : 11188	(0;30] : 7030

We then 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 $t = 3$. We ran the fairlet decomposition in [15] 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 3.6. Colors are named $f1; 2; 3; 4g$ 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 $a > \frac{1}{3}$ for the de nition of a -capped fairness, showing the tree to be unfair.

As in the main body, in Table 3.7, 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 the $\text{ratio}_{\text{value}}$ 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 3.7 Impact of Algorithm 1 on $\text{ratio}_{\text{value}}$ in percentage (mean 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
nal	99.01	0.09	99.41	0.57	99.87	0.28	99.80	0.27	100.00	0.14	99.88	0.30
BANK MULTICOLOR, 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
nal	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 3.8 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

Table 3.8 Impact of Algorithm 1 on ratio, in percentage (mean 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
nal	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
nal	94.99	0.09	95.97	0.07	97.19	0.03	97.98	0.09	98.43	0.04

Table 3.9 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

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 3.9 shows the run time of Algorithm 1 with multiple colors.

3.7 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 nd better upper and lower bounds for fair cost? Are there other important fairness criteria?

Chapter 4

Scaling Average-Linkage via Sparse Cluster Embeddings

4.1 Introduction

Starting with this chapter, the focus is shifted from establishing and refining the mathematical formulation for clustering to designing better, more scalable clustering algorithms. In this chapter we focus on making average-linkage scalable on inputs in Euclidean space using distance/dissimilarity scores. This chapter uses the results in [107]. [Recall from Section 2.2 that average-linkage calculates the average distance over all pairs of points across the clusters. The greater this average distance is, the less similar these two clusters are. Throughout this chapter, this average distance will be referred to as the cluster distance.

Scalability of average-linkage. Agglomerative hierarchical clustering algorithms are known to have scalability issues, especially average-linkage. In fact, introductory guides on hierarchical clustering often state that average-linkage should only be used on small to modestly sized data sets due to the inherent large running time and the sequential nature of the algorithm. This limits the applicability of hierarchical clustering on large datasets [138]. Therefore, there is interest in making average-linkage more scalable.

The fastest known implementation of average-linkage runs in $O(n^2)$ time [116]. This is because the iterative search for the pair of clusters with smallest cluster distance is time consuming. To compute the exact average-linkage output, $\Omega(n^2)$ is intuitively necessarily as the algorithm must inspect all of the $\Theta(n^2)$ pairwise distances between data points to find the first pair to merge. Thus we need better algorithm designs to further improve the running time bound, as the scalability issue arises from the nature of agglomerative algorithms.

Recently, there has been breakthroughs with improving the scalability of average-linkage. The work of Cochez and Nel[56] and Abboud et al[3] broke through the quadratic barrier by relaxing the algorithm. In both works, the idea is to allow the algorithm to iteratively merge clusters that do not necessarily have the smallest average distance. [56] found a sub-quadratic time algorithm for a restricted class of distance functions that includes Jaccard distance, but excludes all ℓ_p-norm distances. Despite being scalable in implementation, the methods in [56] cannot be directly used for any ℓ_p-norm distances. [3] shows that for input with ℓ_1 norm distances, one can also break the quadratic bound by allowing the algorithm to merge clusters with average distance within a factor of $1+\epsilon$ of the average distance between the closest clusters. However, the algorithm design heavily relies on properties of the ℓ_1 norm and the work does not directly apply to other ℓ_p norms[1]. Further, the algorithm emphasizes the near-optimality of every merge performed, leveraging sophisticated data structures to quickly find clusters to merge. The construction and maintenance of such data structures incurs additional computational cost for this adaptation of average-linkage, which is likely to have significant overhead in practice.

Previous work does not directly give a scalable, easily implementable adaptation of average-linkage for Euclidean inputs with general ℓ_p norm distances. Nevertheless, it suggests the exciting possibility of developing such an adaptation with new algorithm designs. The new algorithm design should be supported by similar theoretical guarantees, i.e., it should iteratively choose two clusters to merge, whose distance is up to a constant of the smallest average distance among all clusters. In particular, an intriguing open question is to find an efficient adaptation of average-linkage for the ℓ_2 norm distance, the most popular distance function.

Results. This chapter designs an algorithm that finds an approximation of average-linkage, which applies to ℓ_p norm distances with $p \in [1, 2]$, with sub-quadratic running time guarantees. Empirically, we give a scalable implementation with near-linear running time. Our results are as follows.

- The algorithm runs in near linear time. The algorithm requires $\tilde{O}(dn^{1+\epsilon})$ time and $\tilde{O}(dn^{1+\epsilon})$ space for arbitrarily small $\epsilon > 0$ when the input points belong to d-dimensional Euclidean space[2].

[1] Note it is possible to embed ℓ_p norms into ℓ_1 though an additional data processing step which significantly increase the dimension of the data points.

[2] The $\tilde{O}(\cdot)$ notation hides factors of $\log^c(n)$ for constant $c > 0$.

- The tree constructed is a relaxation of average-linkage that ensures that any pair of clusters merged have average distances within a constant factor of the closest cluster pair.
- We perform experiments with our algorithm and compare it to the most popular implementations of average-linkage such as scikit-learn, sci-py, and fastcluster. These represent the fastest average-linkage packages used by data scientists across industry and academia. Our algorithm: (1) Constructs trees that are similar to what standard libraries achieve. In particular, the trees constructed have almost the same value for a popular hierarchical clustering objective. (2) Achieves asymptotically faster running times than the standard algorithms for large data sets, matching the theoretical guarantees. The running time of our implementation is near linear in the input size, as opposed to the quadratic baselines.
- The algorithm is easily parallelizable and can be used in parallel and distributed environments. There is much previous work on scaling clustering methods such as single-linkage [31]. However, fundamental differences in the problem prevent these methods from extending to average-linkage. In contrast, most recent work on fast average-linkage clustering, such as [3], do not extend easily into parallel and distributed settings.

Cluster embeddings. This chapter introduces a new technique that is of interest beyond average-linkage, named cluster embeddings. This technique maps a given set of clusters into points a slightly higher-dimensional space, such that the pairwise distance between the points roughly preserves the cluster distances. Formally, consider a collection of clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$, a cluster embedding finds k points $v_1, v_2, \ldots, v_k$ where v_i corresponds to C_i. These points exist in some Euclidean space and $\|v_i - v_j\|_p$ should approximate the average distance $\text{Avg}(C_i, C_j)$ up to a constant, i.e., low-distortion [77] with respect to the average ℓ_p distance. Ideally, the points $v_1, v_2, \ldots, v_k$ are located in a space with dimension not much larger than d and the embedding can be computed quickly.

To see the applicability of cluster embeddings, consider a single step of average-linkage. At each step, there is a current collection of clusters $C_1, C_2, \ldots C_k$ and the goal is to find the clusters C_i and C_j that minimizes the average distance between C_i and C_j. That is, clusters whose points are closest on average. Notice that for any pair of clusters it can take quadratic time just to find their average distance depending on the sizes. However, with the embedding in place, we would only need to find the ℓ_p distance between the two embedded points.

The embedding allows us to leverage data mining techniques that apply to points in Euclidean space. Coupling the cluster embedding with Locality-Sensitive Hashing (LSH) functions yields a method to quickly find pairs of clusters with small average distance. Such a task is called *near cluster search*. Given a set of points in Euclidean space, LSH functions partitions the points into "buckets", where points in the same bucket are likely to be closer to each other, thus filtering out potential near neighbors without inspecting all pairwise distances between clusters. Such techniques leverage properties of ℓ_p distances in a Euclidean space and cannot be directly used to do near cluster search, since the cluster distance, defined to be the average distance over all pairs, is a more general metric where LSH may not exist. However, an embedding into Euclidean space makes such a technique applicable for near-cluster searches. Iteratively using the embedding coupled with LSH to find clusters that are close gives a new approach to efficiently approximate average-linkage.

Our cluster embedding has the following properties. We note that Abboud [3] et al. adopted techniques similar to cluster embedding, but these properties set our design apart from theirs.

- The embedding is *sparse*. Clusters of points in d-dimensional Euclidean space are embedded into vectors with at most $cdt + 1$ non-zero entries. Hence only one additional non-trivial dimension is required to store the embedding of each cluster.
- The embedding has *low-distortion* of the cluster distances. The ℓ_p-norm distance between v_i and v_j is guaranteed to approximate the average distance between pairs of points in C_i and C_j. In particular, we prove this is bounded in the worst case. Experimentally, they are nearly the same.
- The embedding is *oblivious*, i.e. there is a function $f : C \to \mathbb{R}^{d'}$ (where d' is the dimension of the embedded space) such that $f(C_i)$, where the computation of $f(C_i)$ depends only on the cluster C_i itself and no other cluster of C. Insertion, deletion, and modification of other clusters doesn't affect $f(C_i)$.
- Constructing the embedding takes *linear time*, $O(nd)$. This is faster than comparing the average similarity between two (large) clusters explicitly (quadratic time).

Chapter Organization: The chapter is organized as follows. We begin by introducing preliminary definitions. Then in Section 4.3, we discuss the technique of cluster embedding - our main technique for speeding up average-linkage. We first showcase the application of cluster embeddings on the problem of near cluster search in Section 4.4. This is the key step in average-linkage. Afterwards, we give an overview of the average-linkage algorithm in Section 4.5. This overview gives a skeleton of the key algorithmic ideas. The detailed

algorithm designs that guarantee strong theoretical performance is defered to 4.5.2. In Section 4.7 we give experimental evaluations.

Related work. There has been consistent interest in finding efficient algorithms to cluster data in large databases [33, 56]. Additionally, there has recently been increased interest in scaling hierarchical clustering algorithms [35, 31, 3, 112, 109]. Kell et al. [91] and Cochez et al. [56] speed-up algorithms by relaxing the criteria on when clusters can be merged. The work of [135] and [97] considered parallelizing hierarchical clustering algorithms in the Massively Parallel Model of computation [85, 20]. Other Spark and MapReduce algorithms have been developed [81, 80]. This prior work on scaling hierarchical clustering focused on other hierarchical clustering algorithms and not average-linkage.

As mentioned, recently there are breakthroughs in designing sub-quadratic adaptations of hierarchical clustering algorithms [56, 3] which do not directly lead to a scalable average-linkage implementation for general ℓ_p norms. Method-wise both chapters used Locality-Sensitive Hashing (LSH) and other related techniques. The work of [3] is closer to our work which focuses on Euclidean inputs using ℓ_2 norm distance. A crucial building block in [3] is to leverage properties of ℓ_1 to construct a sophisticated data structure based on LSH functions, where every cluster is represented as a multi-dimensional point. However their representation of a cluster increases the dimension by a factor of $O(\frac{1}{\epsilon^6}\log^3 n)$ for constant $\epsilon > 0$ and critically uses properties of ℓ_1 that do not hold for other ℓ_p norms. Additionally, the original paper does not include an average-linkage implementation. There remains the question of finding a more concise way of embedding clusters into potentially low-dimensional points and designing a practical adaptation of average-linkage based on this embedding.

Embeddings are widely used and there are many types with different properties. See [70] for reference to different types of embedding. Much prior work has focused on low-distortion embeddings [77]. These embeddings are orthogonal to that considered in this chapter. These are used for reducing dimension and ours is used to represent average cluster distances.

This chapter uses the value objective for hierarchical clustering [59] to evaluate our algorithm. This is the most suitable for the setting we are considering - a point set with pairwise distance metric. Recall that average-linkage is a $\frac{2}{3}$-approximation algorithm for it.

4.2 Preliminaries

Here we define cluster embeddings, average-linkage, near cluster search, hierarchical clustering objective function and Approximate Nearest Neighbor (ANN) search based on Locality-Sensitive Hashing (LSH).

4.2.1 Notation System

We use lower case letters e.g. x, y to denote points in Euclidean space and use subscripts to index their coordinates e.g. x_i, y_i. We will use $\|x\|_p := \left(\sum_i x_i^p\right)^{1/p}$ to denote the ℓ_p-norm of x. $\|x\|$ will be shorthand for the ℓ_2-norm. For a finite set X in d-dimensional Euclidean space $\mathbb{R}^d$, let $D(X)$ and $d(X)$ be the maximum and minimum distances within X, that is, $D(X) := \max_{x,x' \in X} \|x - x'\|$ and $d(X) := \min_{x,x' \in X} \|x - x'\|$. If X is a finite set of points let $m(X) := \sum_{x \in X} x/|X|$ be the centroid of X and let $gm(X) := \arg\min_{y \in \mathbb{R}^d} \sum_{x \in X} \|y - x\|/|X|$ be the geometric median of X.

Average-Linkage: Recall for two disjoint sets A, B, $Avg(A, B) = \frac{1}{|A||B|} \sum_{i \in A} \sum_{j \in B} d(i, j)$ is defined as the cluster distance and should be minimized in every iteration of average-linkage. Here we note that for a collection of clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$, $Avg(\cdot, \cdot)$ induces a metric on $\mathcal{C}$ where we take $Avg(C_i, C_i) := 0$. In our setting we have each $C_i \subseteq \mathbb{R}^d$ and $d(x, y) = \|x - y\|_p$ for some $p \in [1, 2]$.

Cluster Embeddings For Average-Linkage: Let $S \subseteq \mathbb{R}^d$ be a point set and let $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ be a clustering of S. Let $f : \mathcal{C} \to \mathbb{R}^{d'}$ be an embedding of $\mathcal{C}$ into d'-dimensional Euclidean space. We say that f is an embedding of $\mathcal{C}$ into $\ell_p^{d'}$ with distortion a if the following holds for all $C_i, C_j \in \mathcal{C}$: $Avg(C_i, C_j) \le \|f(C_i) - f(C_j)\|_p \le a \cdot Avg(C_i, C_j)$. Ideally we want to embed into a space of dimension d. In lieu of that, we ask for a sparse embedding in a high dimensional space. We say that an embedding is s-sparse if the number of non-zero entries in $f(C_i)$ is at most s for all $C_i \in \mathcal{C}$.

Near Cluster Search: Given a set of clusters $\mathcal{C}$ and a query cluster Q, the near cluster search problem with respect to average-linkage is to find a cluster $C \in \mathcal{C}$ that is the closest to Q in average-linkage. The a-approximate near cluster search problem is to find a cluster $C \in \mathcal{C}$ such that $Avg(Q, C) \le a \cdot \min_{C' \in \mathcal{C}} Avg(Q, C')$.

Approximate Average-linkage: Our algorithms make approximate decisions so we define the notion of approximation here. Sets A, B are a γ-approximate for clustering $\mathcal{C}$ if $Avg(A, B) \le \gamma \cdot \min_{A', B' \in \mathcal{C}} Avg(A', B')$. An algorithm is a γ-approximate if it only merges γ-approximate clusters.

Technical Assumptions: We assume that $\frac{D(S)}{d(S)} = O(\text{poly}(n))$, i.e. the ratio of the maximum to minimum distance is bounded by a polynomial in the number of points. Up to scaling, we may also assume that $d(S)$, the minimum distance, is 1.

4.2.2 Approximate Nearest Neighbors

This subsection gives preliminaries on Approximate Nearest Neighbors (ANN) search. Given a set of points, the ANN technique builds a data structure using these points that

supports efficient queries. Given any point in the given set and a distance upper bound R, an ANN query should either return another point that is approximately within distance R from this point, or return that there are no such points, formally defined as follows.

Definition 4.2.1 (ANN Query [64]). Let S be a set of points and d a distance function on S. Given a new point q, the query either returns a point y with $d(q, y) \leq cR$ or reports that there is no point y' with $d(q, y') \leq R$.

ANN technique is crucial to our scalable average-linkage design. When coupled with the points embedded from clusters, it helps us find candidates for the clusters to merge. The construction of the data structure and the ANN queries show a trade-off between time and computational complexity and nearest point search accuracy. As we allow the approximation ratio, c, to be bigger, the construction and query time will also decline.

Locality Sensitive Hashing (LSH). LSH is the key building block used to construct ANN queries for a given set of points. In our paper, LSH is directly applied to the embedded points to help us find pairs of clusters to merge.

Intuitively, LSH hashes points to values and nearby points are more likely to be hashed to the same value. The space of points can then be partitioned according to the hashed values, each partition named by a bucket. Let X be a metric space with distance function d and U a universe of buckets. We recall the formal definition from [64]. Let $B_S(q, r)$ denote the set of points in S that has distance at most r from point $q \in S$.

Definition 4.2.2 ((r_1, r_2, p_1, p_2)-sensitiveness). A function family $H = \{h : X \to U\}$ is said to be (r_1, r_2, p_1, p_2)-sensitive for $d(\cdot, \cdot)$ if for any $x, y \in X$ and h drawn uniformly at random from H : $d(x, y) \leq r_1 \Rightarrow \Pr_H[h(y) = h(x)] \geq p_1$, and $d(x, y) > r_2 \Rightarrow \Pr_H[h(y) = h(x)] \leq p_2$. We require $p_1 > p_2$ and $r_1 < r_2$.

A family can also be defined by giving a random process which outputs a function from H. LSH families have been constructed for distances induced by ℓ_p norms for all $p \in (0, 2]$ [64]. A hash function $h \in H$ is constructed by the following procedure. Let $r > 0$ be given. Sample a vector $g = (g_i) \in \mathbb{R}^d$, where each g_i is an i.i.d. standard Gaussian, and a real number b uniformly from $[0, r]$. For a point $x \in \mathbb{R}^d$, its hash is $h(x) = \lfloor (\langle g, x \rangle + b)/r \rfloor$. It's known that this family of hash functions is (R, cR, p_1, p_2)-sensitive for any $c > 1$, where p_1, p_2 depend on c. These hash functions can be combined using well-known techniques to amplify collision probabilities and increase accuracy. See [64] for details. Here we summarize this result.

Theorem 4.2.1 (Theorem 1 in [64]). Let H be a (R, cR, p_1, p_2)-sensitive family of hash functions for distance function d. Given a set S of n points there exists an algorithm which constructs a data structure L by sampling multiple times from H which supports (R, c)-ANN queries. The time to construct L and to answer a query from L is dominated by $\tilde{O}(n^{1+r})$ distance computations and $\tilde{O}(n^r)$ distance computations, respectively. Here $r = \frac{\ln 1/p_1}{\ln 1/p_2} \in (0,1)$ depends on c. The bigger c is, the closer r is to 0.

4.3 Sparse Cluster Embeddings for Average Distance

In this section we present our embedding of clusters to points in Euclidean space such that the distance between embedded points approximates the average ℓ_2-distance between clusters. The embedding enables the algorithm to approximately calculate the average distance between any two clusters in $O(d)$ time regardless of their sizes. Later in the discussion on algorithm design, we will show how to use it to facilitate the search for clusters with small average distances. We focus on the case when $p = 2$ and show the following theorem.

Theorem 4.3.1. Given a clustering C of $S \subset \mathbb{R}^d$, there is an embedding $f : C \to \mathbb{R}^{d'}$ into $\ell_2^{d'}$ with distortion a. We have $d' = d + |C|$, $a = 5\sqrt{3}$ and f is $(d+1)$-sparse.

We give a formula which approximates the average-linkage between two clusters A and B. The formula is written in terms of the distance between the centroids of A and B plus two correction terms which only depend on A and B, respectively. This will be used to define the embedding. Then we prove Theorem 4.3.1 using the formula.

Squared Euclidean Distance: We first examine a simpler case to motivate our ideas. Suppose that dissimilarities are given by $d(x, y) = \|x - y\|^2$. Thus $\text{Avg}(A, B) = \sum_{x \in A, y \in B} \frac{\|x - y\|^2}{|A||B|}$. The goal is to express the average-linkage between clusters A, B as the distance between their centroids plus correction terms. Since $\|m(A) - m(B)\|^2$ can potentially be small relative to $\text{Avg}(A, B)$, we need the correction terms to capture the rough variance in each cluster. For a finite set of points X define $\text{Var}(X) := \sum_{x \in X} \frac{\|x - m(X)\|^2}{|X|}$, resembling the definition of "variance" in statistics.

Proposition 4.3.2. For any clusters A and B, $\text{Avg}(A, B) = \|m(A) - m(B)\|^2 + \text{Var}(A) + \text{Var}(B)$.

Before proving this Proposition we first prove the following lemma.

Lemma 4.3.3. For any point y and finite set of points X in Euclidean space $\sum_{x \in X} \|x - y\|^2 = |X|\,\|y - m(X)\|^2 + |X|\,\text{Var}(X)$.

Proof. Rewriting and expanding the square $\|a+b\|^2 = \|a\|^2 + \|b\|^2 + 2\langle a,b\rangle$ we have:

$$\begin{aligned}\sum_{x\in X} \|x - y\|^2 &= \sum_{x\in X} \|x - m(X) + m(X) - y\|^2 \\ &= \sum_{x\in X} \|x - m(X)\|^2 + \sum_{x\in X} \|m(X) - y\|^2 \\ &\quad + \sum_{x\in X} 2\langle x - m(X), m(X) - y\rangle \\ &= |X|\ \mathrm{Var}(X) + |X| \sum_{x\in X} \|y - m(X)\|^2\end{aligned}$$

where the last term on the second line evaluates to 0. □

Proof of [Proposition 4.3.2] Applying Lemma 4.3.3 to the sum $\sum_{y\in B}\sum_{x\in A}\|x-y\|^2$ twice we have:

$$\begin{aligned}\sum_{y\in B}\sum_{x\in A} \|x - y\|^2 &= \sum_{y\in B} |A|\ \|y - m(A)\|^2 + |A|\ \mathrm{Var}(A) \\ &= |A||B|(\|m(A) - m(B)\|^2 + \mathrm{Var}(A) + \mathrm{Var}(B))\end{aligned}$$

and dividing by $|A|\ |B|$ yields the proposition. □

Euclidean Distance: Now we consider dissimilarities given by the Euclidean distance between points. Recall that $\mathrm{Avg}(A,B) = \sum_{x\in A, y\in B} \frac{\|x-y\|}{|A|\,|B|}$. We would like a decomposition similar to Proposition 4.3.2 expressing $\mathrm{Avg}(A,B)$ as the distance between centroids and some correction terms.

Let $\mathrm{Dev}(X) := \sum_{x\in X} \frac{\|x - m(X)\|}{|X|}$ be the average deviation of X from its centroid. It is perhaps intuitive that $f(A,B) := \|m(A) - m(B)\| + \mathrm{Dev}(A) + \mathrm{Dev}(B)$ should suffice based on the results above. However, there are examples where $f(A,B) \not\le \mathrm{Avg}(A,B)$[3]. Instead, we show that the two are always within a constant factor of each other.

Lemma 4.3.4. For any two clusters A and B we have $\mathrm{Avg}(A,B) \le f(A,B) \le 5\,\mathrm{Avg}(A,B)$.

We first prove the following lemmas that better show the connection between $\mathrm{Avg}(A,B)$ and $f(A,B)$.

Lemma 4.3.5. For any cluster X, $\mathrm{Dev}(X) \le 2R_X(gm(X))$.

Proof. By the definition of $\mathrm{Dev}(X)$ and the triangle inequality, $\mathrm{Dev}(X) = \frac{1}{|X|}\sum_{x\in X}\|x - m(X)\| \le \frac{1}{|X|}\sum_{x\in X}(\|x - gm(X)\| + \|gm(X) - m(X)\|)$
$= R_X(gm(X)) + \|gm(X) - m(X)\|$.

[3] Consider the corners of a rectangle with width w and height h, letting A be the left side points and B the right side points. Then $\mathrm{Avg}(A,B) = (w + \sqrt{w^2+h^2})/2 \le w + h = f(A,B)$

Now to complete the proof we claim that $\|gm(X) - m(X)\| \le R_X(gm(X))$. Using the definition of $m(X)$ and subadditivity we have $\|m(X) - gm(X)\| = \sum_{x \in X} \frac{x - gm(X)}{|X|}$ $\frac{1}{|X|}\sum_{x \in X}\|x - gm(X)\| = R_X(gm(X))$. □

Lemma 4.3.6. For any two clusters A, B, $\|m(A) - m(B)\| \le Avg(A,B)$.

Proof. By definition of centroid we have:

$$\begin{aligned} m(A) - m(B) &= \frac{1}{|A|}\sum_{x \in A} x - \frac{1}{|B|}\sum_{y \in B} y \\ &= \frac{1}{|A||B|}\sum_{x \in A}\sum_{y \in B} x - \frac{1}{|A||B|}\sum_{x \in A}\sum_{y \in B} y \\ &= \frac{1}{|A||B|}\sum_{x \in A}\sum_{y \in B} (x - y) \end{aligned}$$

Now applying $\|\cdot\|$ to both sides and then subadditivity of $\|\cdot\|$ completes the proof. □

Lemma 4.3.7. For any two clusters A and B, $R_A(gm(A)) \le Avg(A,B)$.

Proof. Since $gm(A)$ minimizes $R_A(y)$ for any $y \in \mathbb{R}^k$, for all $y \in B$ we have $R_A(gm(A)) \le R_A(y)$. Summing this inequality over all such $y \in B$ we have $|B| R_A(gm(A)) \le \sum_{y \in B} R_A(y) = \sum_{y \in B}\sum_{x \in A} \frac{\|y - x\|}{|A|}$. Dividing both sides by $|B|$ yields the lemma. □

Proof of [Lemma 4.3.4] We start by proving the lower bound. By the triangle inequality, for any $x \in A$ and $y \in B$ we have $\|x - y\| \le \|x - m(A)\| + \|m(A) - m(B)\| + \|y - m(B)\|$. Summing this inequality over all $x \in A$ and all $y \in B$ then dividing by $|A| \cdot |B|$ yields $Avg(A,B) \le f(A,B)$. For the upper bound we have the following sequence of inequalities: $\|m(A) - m(B)\| + Dev(A) + Dev(B) \le Avg(A,B) + 2R_A(gm(A)) + 2R_B(gm(B)) \le 5\,Avg(A,B)$.

The first inequality follows from Lemmas 4.3.6 and 4.3.5, and the second inequality follows from applying Lemma 4.3.7 to the last two terms. □

The lower bound $Avg(A,B) \le f(A,B)$ is a simple application of the triangle inequality. To show the upper bound we must relate $Dev(A)$ and $Dev(B)$ to $Avg(A,B)$. Recall that for a cluster X, its geometric median $gm(X)$ minimizes $R_X(y) := \frac{1}{|X|}\sum_{x \in X}\|y - x\|$. The value of $R_X(y)$ is the average distance of points in X to y. We show that for any two clusters A, B, both $R_A(gm(A))$ and $R_B(gm(B))$ lower bound $Avg(A,B)$. Then we show that for any cluster X, $Dev(X) \le 2R_X(gm(X))$. Combining these two facts with $\|m(A) - m(B)\| \le Avg(A,B)$ implies the upper bound.

Constructing the Embedding: We now define the embedding and prove Theorem 4.3.1. Let $C = \{C_1, C_2, \ldots, C_k\}$ be a clustering of a point set $S \subset \mathbb{R}^d$. We define the embedding

$f : C \to \mathbb{R}^{d+|C|}$ for each $C_i \in C$ as follows.

$$f(C_i) := \sqrt{3}(m(C_i), 0, \ldots, \underbrace{\mathrm{Dev}(C_i)}_{d+i\text{'th coordinate}}, \ldots, 0) \tag{4.1}$$

In other words we set the first d coordinates to be the centroid of C_i, the $d+i$'th coordinate to be $\mathrm{Dev}(C_i)$ and all other coordinates to be 0. Observe that the embedding is oblivious, meaning that computing $f(C_i)$ only depends on the cluster C_i and none of the other clusters. Given this embedding we have that the distance between the embedding of clusters C_i and C_j, $\|f(C_i) - f(C_j)\|$, is equal to

$$\sqrt{3}\sqrt{\|m(C_i) - m(C_j)\|^2 + \mathrm{Dev}(C_i)^2 + \mathrm{Dev}(C_j)^2} \tag{4.2}$$

To finish the proof, we use some easy to verify inequalities.

Proposition 4.3.8. For all $a, b, c \in \mathbb{R}_+$, we have: $(1/\sqrt{3})(a+b+c) \leq \sqrt{a^2+b^2+c^2} \leq a+b+c$.

Proof of Theorem 4.3.1. Let $C = \{C_1, C_2, \ldots, C_k\}$ be a clustering of S and consider the embedding $f : C \to \mathbb{R}^{d+|C|}$ defined by (4.1). The distance between embedded clusters $f(C_i)$ $f(C_j)$ is given by (4.2). Combining this with the respective lower/upper bounds from Proposition 4.3.8 and Lemma 4.3.4, we have $\|f(C_i) - f(C_j)\| \leq f(C_i, C_j) \leq \mathrm{Avg}(C_i, C_j)$ and $\|f(C_i) - f(C_j)\| \geq \sqrt{3} f(C_i, C_j) \geq 5\sqrt{3}\,\mathrm{Avg}(C_i, C_j)$. Finally, f being $(d+1)$-sparse follows directly from (4.1). □

We note that [64] has shown that the LSH scheme is known to work for any ℓ_p-norm where $p \in (0, 2]$. One can easily see that the following Proposition, similar to Proposition 4.3.8 holds:

Proposition 4.3.9. For all $a, b, c \in \mathbb{R}_+$, and $p \geq 1$, $\frac{1}{3^{1-\frac{1}{p}}}(a+b+c) \leq (a^p + b^p + c^p)^{\frac{1}{p}} \leq a+b+c$.

The main idea for the proof of Theorem 4.3.1 can be easily extended to work for general ℓ_p-norm where $p \in [1, \infty]$, resulting in the following Corollary.

Corollary 4.3.9.1. Given a clustering C of $S \subseteq \mathbb{R}^d$, there exists a constant $\alpha \geq 1$ and a mapping $f : C \to \mathbb{R}^{d'}$ such that for all $A, B \in C$, $\mathrm{Avg}(A, B) \leq \|f(A) - f(B)\| \leq \alpha\,\mathrm{Avg}(A, B)$. In particular, we have $d' = d + |C|$ and $\alpha = 5 \cdot 3^{1-\frac{1}{p}}$.

4.4 Near Cluster Search

In the section we consider the key subproblem solved in average-linkage to showcase the cluster embedding idea. At each step, average-linkage finds the pair of cluster that are the closest. We abstract this to the following *near cluster search* problem, an extension of the ANN query to clusters instead of points. The goal is to find an approximately nearest cluster in a clustering $\mathcal{C}$ to a query cluster Q with respect to the average distance metric. This subproblem is of its own interest, because data scientists often seek to find pairs of clusters that are very similar.

The naive approach for finding the cluster $C \in \mathcal{C}$ minimizing $\mathrm{Avg}(Q, C)$ requires $O(d|Q| \sum_i |C_i|)$ time. Our cluster embedding can be used to give an asymptotically faster but approximate algorithm for this task, taking $O(d(|Q| + \sum_i |C_i|))$ time.

Theorem 4.4.1. *There exists a data structure supporting $\sqrt{3}$ approximate near cluster search queries in time $O(d(|\mathcal{C}| + |Q|))$. Construction takes time $O(d(|Q| + \sum_i |C_i|))$.*

Proof. Let $\mathcal{C}' = \mathcal{C} \cup \{Q\}$ and let f be the embedding given by Theorem 4.3.1 for $\mathcal{C}'$. Start by computing $f(C_i)$ for each $C_i \in \mathcal{C}$, taking time $O(d \sum_i |C_i|)$. Note that f is oblivious so this step can be done without knowing Q. The data structure stores the embedded points. To answer a query Q, compute $f(Q)$ and then compute the distance from $f(Q)$ to $f(C_i)$ for each $C_i \in \mathcal{C}$ and return C_i with the smallest embedded distance. Since f has distortion $\sqrt{3}$ the approximation guarantee follows. Since f is $(d+1)$-sparse, answering the query takes time $O(d|Q| + d|\mathcal{C}|)$. □

Theorem 4.4.1 speeds up near cluster search for any cluster Q enormously, as is shown in Section 4.7. Further, coupling the embedding with ANN techniques allows us to query in sublinear time, while only losing an additional constant factor in the approximation.

Using this, we can construct the point set S' consisting of the embedded points $f(C_i)$ for all $C_i \in \mathcal{C}$. We apply the above to obtain a data structure L supporting efficient approximate near neighbor search queries in the embedded space. The sparse nature of the embedding also implies that L can be computed very efficiently in practice. When given query Q, we compute $f(Q)$ and then apply the algorithm from Theorem 4.2.1 to process the query. We get the following result.

Theorem 4.4.2. *For any constant $\varepsilon > 0$, there exists a data structure which supports $\sqrt{3}c$-approximate near cluster search queries in time $\tilde{O}(d(|\mathcal{C}|^r + |Q|))$. Constructing the data structure takes time $\tilde{O}(d(\sum_i |C_i| + |\mathcal{C}|^{1+r}))$. Here $r \in (0,1)$ is a constant depending on c.*

4.5 Fast Approximate Average-Linkage

The main application of our cluster embedding is to get fast approximate algorithms for average-linkage. In this section we give the main structure for our fast implementation of average-linkage that highlights the key ideas. The idea is to embed the current set of clusters into points and then partition the embedded points into buckets (sets) using LSH. The number of buckets is not fixed. We can use the hash family described in Section 4.4 for this.

4.5.1 A Sketch for ANN-based Average-Linkage

Before diving deeper into the full algorithm, we provide a sketch that can help readers understand the key designs at an intuitive level. Say we want to merge all pairs of clusters with average distance approximately d. The algorithm draws hash function h randomly from H, where H is the LSH family given in [64], and applies it to $f(A)$ for each cluster $A \in C$. Let the i'th bucket be $B_i = \{A \in C \mid h(A) = i\}$[4]. Suppose bucket B_i is small: $|B_i| \leq O(n^r)$ for some $r \in (0,1)$. We run vanilla average-linkage inside B_i to the point where there is no pair of clusters with average distance smaller than $(1+\epsilon)d$. Here we crucially treat each cluster in a bucket B_i as a single point via the cluster embedding, which allows us to replace computing the average-linkage between pairs of clusters with a distance computation in the embedding. This takes $\tilde{O}(dn^{2r})$ time. Moreover, assuming all buckets are small, the cost over all buckets is $\tilde{O}(dn^{1+r})$. We call this the "local merging" step.

Once all buckets have completed their "local merging" step, the algorithm repeats the above steps with the given value of d for $O(\log n)$ iterations. Provably, no two clusters will have average-linkage $(1+\epsilon)d$ with high probability. At this point the algorithm recurses with $d \leftarrow (1+\epsilon)d$. The number of such iterations is bounded by the number of distances considered between the minimum and maximum possible distances, which is $O(\log n)$ by our technical assumption. See Algorithm 2 for pseudocode. Given the above discussion, we see that the running time should be $\tilde{O}(dn^{1+r})$, near linear time for small constant r.

There are two unaddressed challenges. First, what should be done when there are buckets B_i with $|B_i| > n^r$. Second, maintaining the embedding can be expensive if done in a naive manner. We briefly address these challenges here, but note that a variant of the above procedure works in experimentation.

Challenge 1 - Large Buckets: In the worst case, it is possible that LSH can produce buckets of size larger than n^r, resulting in a poor running time for the local merges. Notably, empirically most buckets have small sizes after LSH. For the few buckets whose sizes are

[4] We use $h(A)$ as a shorthand for $h(f(A))$ since we will always be implicitly working in the embedded space

Algorithm 2 Fast Approximate Average-Linkage Algorithm

```
1: FastAverageLinkage(S, α, ε):
2: C ← {{x} | x ∈ S} {Make leaf clusters}
3: for k = 1, 2, ..., Q(log n) do
4:    δ ← (1 + ε)^(k−1)
5:    C_final ← ∅ {Tracks clusters that won't get merged again for this value of δ}
6:    for t = 1, 2, ..., Q(log n) do
7:       h ← random hash function from H
8:       Compute h(A) for each A ∈ C
9:       Let B_i = {A ∈ C | h(A) = i}
10:      if |B_i| ≤ O(n^r) for all i then
11:         for each B_i do
12:            Run vanilla average-linkage on B_i until no merges w/ average-linkage
               ≤ α(1 + ε)δ
13:         end for
14:         Update the embeddings
15:      else
16:         C, C_final ← RobustMerging(C, C_final, δ, α, ε)
17:      end if
18:   end for
19:   C ← C_final
20: end for
21: Return the hierarchical clustering found.
```

large, we use the "RobustMerging" procedure that utilizes the ANN search as mentioned in Theorem 4.2.1. For any cluster, this technique only checks $\tilde{O}(k n^r)$ clusters that got hashed into the same bucket and gives theoretical performance guarantees.

Challenge 2 - Updating the Embedding: If one wants to update the embedding after a single merge, this requires computing the distance of each point to the new centroid. This is potentially a complex and expensive operation we wish to avoid. To do so we carefully make additional merges before updating the embedding and applying LSH again. This is done to ensure that enough clusters have been involved in a merge so that the total cost due to updating the embedding is small. In practice, we use reservoir sampling to reduce the cost of updating the embedding by approximating it using a set of $\Theta(d \log n)$ samples maintained during the merges.

4.5.2 Robust Merging

In this section we unveil the robust merging subroutine ("RobustMerging") omitted in Algorithm 2. We show that the robust merging procedure tackles the main challenges

mentioned and is key to ensuring that the algorithm runs in near-linear time in the worst-case and approximates average-linkage.

In Algorithm 3, $a = 5 + \sqrt{3}$ is the distortion factor in Theorem 4.3.1. The goal of this section is to show the following theorem.

Theorem 4.5.1. With high probability, Algorithm 2 along with Algorithm 3 is a $(3+\varepsilon)$-approximate for average-linkage and runs in time $\tilde{O}(dn^{1+\rho})$ for some constant $\rho \in (0,1)$.

The following lemma will be quite useful in the analysis and will guide the development of the merging procedure. The proof is easy and follows from a simple averaging argument.

Lemma 4.5.2. For a cluster A, if $\text{Avg}(A,B) \geq d$ and $\text{Avg}(A,C) \geq d$, then $\text{Avg}(A, B \cup C) \geq d$.

In other words this lemma states that if after some merge the minimum average distance between clusters is at least d, we are guaranteed that it will not fall below d during any future merges. Intuitively, we want to merge clusters an increasing over time. This lemma ensures that we do not undo progress.

Recall that one of the main challenges discussed is that LSH may produce large buckets. To handle this, we employ the Approximate Near Neighbor (ANN) search technique from [64]. Recall De nition 4.2.1 and Theorem 4.2.1 from earlier in the paper. We will use these tools in our analysis.

The robust merging procedure leverages (R, cR)-ANN queries in the following way. First, we embed the clusters into points. By Theorem 4.3.1, this preserves the average-linkage up to the constant distortion factor a. We then build the data structure referenced above and use it to query every cluster.

The robust merging procedure is given in Algorithm 3. This can be described as follows. We say two clusters are "neighbors", if querying one of them returns the other. For each $A \in C$, let $Q(A)$ denote the set of its neighbor clusters. If $Q(A) = \emptyset$, with high probability A has average distance of at least $(1+\varepsilon)d$ from any other cluster. We remove A from C and add it to C_{final}. Otherwise, if set $Q(A)$ has valid clusters that are not yet merged with any other cluster, we iteratively merge A with these clusters in any arbitrary order. However, if set $Q(A)$ is non-empty but all clusters are already merged with their other neighbors, we pick any neighbor from $Q(A)$ and merge A with the current cluster containing it.

We give intuition for why the algorithm is approximate and has good running time. The formal proof is given in the next section.

Algorithm 3 Pseudocode for Robust Merging Procedure

```
1: RobustMerging(C, ℓ, d, a, e)
2: L ← (a(1+e)d, c)-ANN data structure from Theorem 4.2.1 {Construct it for point set
   {f(A) | A ∈ C}}
3: for A ∈ C do
4:     Q(A) ← ∅ {Will be set of discovered "neighbors" of A}
5: end for
6: for A ∈ C do
7:     B ← result of querying f(A) in L
8:     if B is a cluster then
9:         Q(B) ← Q(B) ∪ {A}, Q(A) ← Q(A) ∪ {B}
10:    end if
11: end for
12: for A ∈ C such that Q(A) = ∅ do
13:    Drop A from C and add it to C_final. Delete f(A) from L. {A won't be considered for
       merging again at the current d}
14: end for
15: for A ∈ C such that Q(A) is not empty do
16:    if Q(A) has clusters that are not yet merged with any other cluster then
17:        Merge all unmerged clusters in Q(A) ∪ {A} with A in any order
18:    else
19:        Let B be arbitrary cluster in Q(A)
20:        Merge A with the cluster containing B
21:    end if
22: end for
23: Remove all the invalid clusters from C and add all the new clusters into C
24: Return C, C_final
```

4.6 Algorithm Analysis: Approximation Ratio and Running Time

4.6.1 Running Time Analysis for Algorithm

To show that the algorithm is efficient we need to show that in every round of robust merging makes reasonable progress. In particular, we show that in the total number of clusters remaining in C shrinks by a constant factor. For fixed d, every time robust merging is executed, any cluster in C is either put into C_{final} or merged with another cluster. Therefore, $|C|$ shrinks by at least $\frac{1}{2}$. Thus for the same threshold d, after $Q(\log n)$ rounds of robust merging there will be no pairs of (A, B) such that $Avg(A, B) \leq (1+\epsilon)d$. This is formalized in the next section.

First we give a lemma showing there exists an estimate $f(A,B)$ $Avg(A,B)$ update formula which preserves the approximation ratio for $Avg(A,B)$.

Lemma 4.6.1. Let $f(A,B)$ be estimates such that $Avg(A,B) \le f(A,B) \le a\,Avg(A,B)$ for all $A,B \subseteq C$. Suppose we merge $A,A' \subseteq C$ and set $f(A\cup A',B) = \frac{|A|}{|A|+|A'|} f(A,B) + \frac{|A'|}{|A|+|A'|} f(A',B)$, then $Avg(A\cup A',B) \le f(A\cup A',B) \le a\,Avg(A\cup A',B)$ for all $B \subseteq C \setminus \{A,A'\}$.

Lemma 4.6.1 follows easily by an averaging argument using $Avg(A\cup A',B) = \frac{|A|Avg(A,B)+|A'|Avg(A',B)}{|A|+|A'|}$. This implies that we can simulate vanilla average-linkage within each bucket in Algorithm 2, using the update formula given in this lemma to simulate the updates in average-linkage. Combining this with current average-linkage implementations, we can approximately simulate the vanilla average linkage algorithm in bucket B_i in time $\tilde{O}(|B_i|^2)$, until there are no pairs of clusters (A,B) such that $f(A,B) \le a(1+\varepsilon)d$. Thus, there are no pairs of (A,B) such that $Avg(A,B) \le (1+\varepsilon)d$.

We will now show the run time.

Theorem 4.6.2. Algorithm 2 has a run time of $\tilde{O}(dn^{1+\rho})$ for some constant $\rho \in (0,1)$.

Proof. Let $\rho = \ln(1/p_1)/\ln(1/p_2) \in (0,1)$, where p_1, p_2 are the probabilities for the LSH family in [64]. Note that it is possible to take p_1, p_2 as constants. Since the $\tilde{O}(\cdot)$ notation hides factors of $O(\log(n))$, it suffices to analyze the cost of the work done in the innermost loop. This is because by definition of the algorithm, the steps in the innermost loop are repeated at most $O(\log^2 n)$ times. We first show that trivially, the time spent on updating the embedding $f(\cdot)$ is $\tilde{O}(n)$. This comes from the fact that updating centroid takes $O(1)$ time every merge, and there are $n-1$ merges in total. For deviation term, we do $\log^2(n)$ rounds of merging. During every round of merging, every point in the original cluster is calculated in the deviation terms only once, thus giving $\tilde{O}(n)$ time spent on updating. Thus we want to bound the cost of running a vanilla algorithm on each bucket when all bucket sizes are bounded by $O(n^\rho)$ and the cost of running the robust merging procedure.

Suppose that we have buckets B_i with $|B_i| \le O(n^\rho)$ for all i. Note that $\sum_i |B_i| = |C|$ since the buckets partition the current set of clusters Then by Lemma 4.6.1 running a vanilla average-linkage algorithm on each bucket costs at most $O(|B_i|^2)$ per bucket. Thus the overall run time cost is $O(\sum_i |B_i|^2) = O(n^\rho \sum_i |B_i|) = O(n^\rho |C|) = O(n^{1+\rho})$ since $|C| \le n$.

Now consider the cost of running the robust merging procedure. By Theorems 4.3.1 and 4.2.1 the data structure can be constructed in time $\tilde{O}(|C|^{1+\rho}) = \tilde{O}(n^{1+\rho})$. Again by Theorem 4.2.1, querying each cluster takes time $\tilde{O}(|C|^{1+\rho}) = \tilde{O}(n^{1+\rho})$ distance computations in total. Finally, the merging step takes time $O(|C|) = O(n)$, yielding an overall cost of $\tilde{O}(n^{1+\rho})$ for the robust merging procedure. □

4.6.2 Approximation and Correctness

Now we move to analyzing the correctness of our algorithm. This encompasses two different claims, one concerning how well the algorithm approximates average-linkage and another showing that the algorithm always produces a valid hierarchical clustering tree. The second claim is important since it is not immediately obvious from the de nition of Algorithm 2. In particular we need to show that the algorithm will perform enough merges and does not terminate before a tree is completed. The following theorems formalize these claims.

Theorem 4.6.3. Assuming the minimum pairwise distance to be lower-bounded by 1, Algorithm 2 is $O(1)$-approximate for average-linkage with high probability.

Theorem 4.6.4. With high probability, Algorithm 2 returns a valid hierarchical clustering tree upon termination.

Theorem 4.6.3 establishes the fact that Algorithm 2 is $O(1)$-approximation for average-linkage. Note that Lemma 4.6.7 only establishes this for the robust merging procedure, so to complete this analysis we need to look at Algorithm 2, which uses the robust merging procedure. The proof of Theorem 4.6.4 is also tied to the proof of Theorem 4.6.3, so we will do the analysis for both here.

To prove the above theorems we make use of the following key lemmas. These guarantee that for each threshold value d, by the end of the $O(\log n)$ rounds of merging, there are no good pairs left to merge for the current threshold, and every merge we do has average distance close to $(1+\epsilon)d$.

Lemma 4.6.5. For threshold d after merging with high probability, there are no pair of clusters with distance at most $(1+\epsilon)d$ left.

Lemma 4.6.6. For a threshold value d, every merge done by Algorithm 2 has average distance at most $4\alpha(1+\epsilon)d$.

Before proving Lemmas 4.6.5 and 4.6.6, we rst show that they directly give us Theorems 4.6.3 and 4.6.4. To simplify the language in our arguments, we define $(1+\epsilon)d$-good clusters as follows.

De nition 4.6.1. During any stage of Algorithm 2, a cluster is called $(1+\epsilon)d$-good, if there exists one cluster in the current clustering with average distance at most $(1+\epsilon)d$. We say a pair is $(1+\epsilon)d$-good if the average distance is at most $(1+\epsilon)d$.

Notice that with the de nition of $(1+\epsilon)d$-goodness, Lemma 4.6.5 is equivalent to claiming that by the end of merging there are no $(1+\epsilon)d$-good points left, or no $(1+\epsilon)d$-good pairs left. We are now ready to prove the theorems.

Proof of [Theorem 4.6.3 and 4.6.4] For Theorem 4.6.3, given any threshold d, take any merge $(A;B) \to A \cup B$ done in any round for d, by Lemma 4.6.6 we have $Avg(A;B) \le 4ca(1+e)d$. However, $\min_{\{S_1;S_2\} \subseteq C} Avg(S_1;S_2) > d$ by Lemma 4.6.5, since all LSH rounds for threshold value $d=(1+e)$ are over. Therefore, the approximation ratio is at most $4ca(1+e)$ for all merges.

For Theorem 4.6.4, since with high probability it eliminates all $(1+e)d$-good points for threshold d, and within $O(\log n)$ number of thresholds we have investigated all the possible average-distance values since it must be at most $\max_{\{i;j\} \subseteq S} D(i;j)$ and therefore merged all the pairs. □

Now we prove Lemmas 4.6.5 and 4.6.6.

Before proving Lemma 4.6.6 we first its simplied version. That is, the quality of every merge is high in the Robust Merging Algorithm 3.

Lemma 4.6.7. Every cluster $A \in C$ such that $Q(A)$ is non-empty is merged with some other clusters, and every merge done by Algorithm 3 has average distance at most $4ca(1+e)d$.

Proof. In any iteration of robust merging, a cluster A, if not merged with other clusters yet, is merged with its unmerged neighbors $Q(A)$, or it gets added to a set of clusters which are already merged. We call the first type of cluster "core" clusters.

Pick any core cluster A in C at the beginning of Algorithm 3. Any other cluster C, if not put into C_{final}, must be merged with one such cluster. We show that by the end of robust merging, for every cluster B that is merged with core A, $Avg(A;B) \le 2ca(1+e)d$.

Pick any cluster B that is in the same cluster with A at the end of Algorithm 3. By Theorem 4.3.1, we have $Avg(A;B) \le \|f(A) - f(B)\|$. If $B \in Q(A)$, we have $\|f(A) - f(B)\| \le ca(1+e)d$ since B is returned by doing $(ca(1+e)d;c)$-NN query on A. Otherwise, it must be the case that some other cluster $C \in Q(B)$ is in the same cluster as A. If $C \in Q(A)$, when robust merging comes to C, it would have merged C with B since $B \in Q(C)$ and B hasn't been merged with any cluster yet, contradicting the assumption that B is merged with A. Therefore $C \notin Q(A)$, and $Avg(A;B) \le \|f(A) - f(B)\| \le \|f(A) - f(C)\| + \|f(C) - f(B)\| \le 2ca(1+e)d$. Now by triangle inequality any two clusters marked with A has average-linkage at most $4ca(1+e)d$. □

The proof of Lemma 4.6.6 is a simple application of Lemma 4.6.7 combined with the fact that the algorithm does average-linkage on the embedded clusters until the minimum average distance reaches the threshold $(1+e)d$. So in both robust merging and average-linkage merging, the pairs we merge are always $4ca(1+e)d$-good. The rest of this section will be devoted to proving Lemma 4.6.5.

In each big WHILE loop of LSH, Algorithm 2 can go to vanilla average-linkage if all buckets are small, and to subroutine RobustMerging if we have at least one big bucket. For any round of LSH, we have the following lemma.

Lemma 4.6.8. Every round of LSH that ends up with doing average-linkage within each bucket has at least constant probability p_0 of reducing the number of $(1+\epsilon)d$-good pairs by a constant of $\frac{p_1}{10}$, and $p_0 \geq \frac{9}{10}p_1$.

Proof. Let's focus on one round of LSH, suppose that before LSH we have N number of $(1+\epsilon)d$-good pairs. Regardless of whether average-linkage or RobustMerging takes place after LSH, we use E_0 to denote the event, where at least $\frac{p_1}{10}N$ $(1+\epsilon)d$-good pairs collide. Let $p_{collide}$ be the probability of this occurring. Let $N_{collide}$ denote the number of $(1+\epsilon)d$-good pairs that collide during the LSH. Since every good pair has probability at least p_1 of colliding, we have $E[N_{collide}] \geq p_1 N$ by linearity of expectation. So

$$p_1 N \leq E[N_{collide}] \leq (1 - p_{collide})\frac{p_1}{10}N + p_{collide} N$$
$$\frac{p_1}{10}N + p_{collide}N$$

This yields $p_{collide} \geq \frac{9}{10}p_1$. Note that the event E_0 happens with probability at least $p_{collide}$, independent of the location of the points and the merges we have done before. □

We have established the fact that every single round of LSH has some constant probability p_0 of getting a constant portion of good pairs to collide. Here we quote a version of Chernoff's bound:

Theorem 4.6.9 (Lower Chernoff Bound). Let $X_1, X_2, \ldots, X_t$ be a collection of independent Bernoulli RV's with $Pr[X_i = 1] = p_i$. Let $X = \sum_i X_i$ and $m = \sum_i E[X_i]$. Then for all $\delta \in (0,1)$, $Pr[X < (1-\delta)m] \leq \exp\left(-\frac{\delta^2 m}{2}\right)$.

By performing $Q(\frac{1}{p_0}\log(n))$ rounds of LSH and using Theorem 4.6.9, with high probability E_0 happens $Q(\log(n))$ times during these rounds. We focus only on the LSH rounds where E_0 happens. During these rounds, Algorithm 2 either goes to average-linkage or robust merging. At least one of them will be repeated $\Omega(\log(n))$ times.

For average-linkage, we prove the following lemma. Intuitively, it guarantees that the total number of good pairs will go down by a constant factor.

Lemma 4.6.10. Suppose in the t^{th} round of LSH event E_0 happens. Let N_t denote the number of $(1+\epsilon)d$-good pairs before and N_{t+1} denote the number of $(1+\epsilon)d$-good pairs after. If Algorithm 2 goes to average-linkage, then $N_{t+1} \leq \frac{9}{10}N_t$. If it goes to robust merging, then $N_{t+1} \leq N_t$.

	Blog				Shuttle				Covertype				Higgs (L)				Susy (L)			
k	Embed		Base		Embed		Base		Embed		Base		Embed		Base		Embed		Base	
128	0.17	0.003	25.1	28.2	0.09	0.002	34.2	44.7	0.11	0.001	32.5	9.1	0.11	0.001	22.3	5.4	0.10	0.001	24.5	10.5
256	0.17	0.002	9.5	9.7	0.09	0.001	18.6	17.8	0.11	0.002	10.5	4.5	0.11	0.001	11.5	2.2	0.10	0.002	11.2	4.2
512	0.17	0.005	14.2	33.4	0.09	0.002	6.3	6.9	0.12	0.004	6.3	2.4	0.11	0.002	4.8	1.9	0.10	0.002	4.9	3.3
1024	0.18	0.01	3.2	3.4	0.09	0.002	2.0	1.8	0.12	0.002	2.9	1.2	0.12	0.002	3.3	0.7	0.11	0.002	2.9	1.0
2048	0.18	0.02	1.8	2.8	0.11	0.012	0.79	0.5	0.13	0.02	1.3	0.6	0.13	0.01	1.4	0.5	0.11	0.002	1.0	0.6
4096	0.19	0.003	0.82	0.3	0.12	0.01	0.45	0.21	0.14	0.003	0.62	0.3	0.15	0.01	0.53	0.2	0.13	0.002	0.52	0.4

Table 4.1 Run time results for Near Cluster Search in seconds. We report the results as m s, where m is the average run time at each value of k and s is the standard deviation. Note that our technique consistently has a small run time with minuscule variance compared to the highly variable baseline.

Proof. After average-linkage, since we have merged the clusters until the minimum average distance for $f(\cdot)$ hits a $(1+\epsilon)d$, the true minimum average distance is more than $(1+\epsilon)d$. So there remains no $(1+\epsilon)d$-good pairs in all the buckets.

If we can somehow prove when we merge two clusters in the same bucket, the number of $(1+\epsilon)d$-good pairs across the buckets does not go up, we can safely conclude that $N_{t+1} \leq N_t - N_{collide} \leq \frac{9}{10}N_t$. Suppose average-linkage chooses to merge clusters A and B. For any cluster C in other buckets, if $Avg(A \cup B, C) \leq (1+\epsilon)d$, we must have $\min\{Avg(A,C), Avg(B,C)\} \leq (1+\epsilon)d$, so at least one of the two pairs (A,C) and (B,C) are good. Now we merge A and B and only one good pair is produced, therefore the number of $(1+\epsilon)d$-good pairs across the buckets will not increase if we do average-linkage within each bucket. For the same reason, if Algorithm 2 does robust merging, we also have $N_{t+1} \leq N_t$. Thus Lemma 4.6.10 is proved. □

Recall the following lemma from previous subsection:

Lemma 4.6.11. Fix any threshold d and let C^t be the set C in Algorithm 3 after t iterations. We have $|C^{t+1}| \leq \frac{1}{2}|C^t|$.

Intuitively, in every LSH such that at least $\frac{p_1}{10}$ portion of $(1+\epsilon)d$-good pairs collide, either buckets are small, in which case the number of $(1+\epsilon)d$-good pairs will shrink by a constant factor with constant probability $p_{collide}$, or there is at least one bucket that's big, in which case the number of candidates for $(1+\epsilon)d$-good clusters will shrink by a constant factor with high probability. Putting Lemma 4.6.10 and 4.6.11 gives us Lemma 4.6.5.

4.7 Experiments

We evaluate the performance of the cluster embedding for two applications: near cluster search and average-linkage hierarchical clustering. The goal is to establish the following:

- Our algorithms which use cluster embeddings are scalable for both applications. Their running time drastically outperforms standard implementations on large data sets.

Dataset	Approx. ratio	Embedding ratio
Shuttle	1.003	1.072
Blog	1.009	1.053
Covertype	1.022	1.059
Susy (L)	1.019	1.032
Higgs (L)	1.012	1.016

Table 4.2 Approximation Results for Cluster Search. Note that these ratios tend to be very close to 1, indicating the accuracy of our algorithm.

Algorithm	Blog 32768	Blog 52397	Shuttle 32768	Shuttle 43500
scipy	221.76	732.43	125.27	307.43
sklearn	247.88	754.16	127.55	305.45
fastcluster	245.49	689.19	96.43	228.02
Our method	129.88	205.70	48.98	78.76

Table 4.3 Run time results for average-linkage clustering. Each column is labeled by the data set and input size considered. Our algorithm consistently sees a speed-up at moderate input sizes over all baselines tested.

- The running time of our average linkage implementation is only slightly super-linear in the input size.
- Distances in the embedding closely approximate the average distance for all data sets.
- Our algorithm for near cluster search returns a cluster with approximately the smallest average distance. In fact, it returns the exact best cluster at least 90% of the time.
- Our average-linkage hierarchical clustering algorithm closely approximates the standard average-linkage algorithm. The vast majority of the merges made by the algorithm have average distances below a factor 1.5 of the minimum average distance between clusters.
- The objective value of trees produced by our new average-linkage algorithm has negligible loss compared to baseline algorithms.

Experiment Specifications: We implemented our algorithms in Python and used 3 baseline implementations of average linkage available in the SciPy, Scikit-Learn, and Fastclustering libraries. Experiments utilized Google Cloud Platform [1] virtual machines (VMs), specifically n1-highmem-4 type VMs, each with 4 virtual CPUs and 26 GB of memory.

We further compare to the work of [3]. For this comparison, we used the ℓ_1-norm distance, a requirement of their algorithm. Our method and baselines were superior to the implementation of [3] in all experiments. While their algorithm has near linear run time like ours, their method runs in time $O(\frac{1}{\epsilon^6} n^{1+\epsilon} \log^3 n)$. The $\frac{1}{\epsilon^6} \log^3 n$ factor makes this algorithm slow in practice.

Datasets: We use datasets from the UCI ML repository [104] : Shuttle, Blog, Covertype, Susy, and Higgs. Number of dimensions range from 28 to 281. Datasets are listed in Table 4.4. Some datasets contained extra features which are a function of the other features. These were preprocessed to create two datasets: “Large” (L) with all features and "Small" (S) without the extra features. If the dataset contained a class column for supervised learning, that column was removed. Euclidean distance was used as the dissimilarity between datapoints.

Dataset	Number of Points	Number of Features	Pre-processed?
Shuttle	43500	8	N
Blog	600021	281	N
Covertype	581012	54	Y
Susy (L/S)	5000000	18/8	Y
Higgs (L/S)	11000000	28/21	Y

Table 4.4 Data sets tested

Implementation Details: For near cluster search, we implemented the algorithm based on Theorem 4.4.1. Our implementation of average-linkage is similar to Algorithm 2 but include a few minor changes for practical efficiency.

Merging procedure for large buckets: in practice, if the partition induced by a round of LSH has a large bucket with more than $\sqrt{n}$ points, we split it into several small buckets of size at most $\sqrt{n}$. Then we run average linkage within each small bucket.

Reservoir sampling to approximate deviation terms: we estimate the deviation $Dev(A)$ term in the embedding $f(A)$ by sampling a small set of points uniformly from the points in this cluster: if x is a uniformly random point from A, then $E[\|x - m(A)\|] = Dev(A)$. A set of $Q(\log n)$ samples is maintained for every cluster, and updated using reservoir sampling when merging two clusters. This makes it easy maintain estimates of the deviation term.

Algorithm	Covertype 65536	Covertype 262144	Higgs (L) 65536	Higgs (L) 262144	Higgs (S) 65536	Susy (L) 65536	Susy (L) 262144	Susy (S) 65536
fastcluster	584.06	-	667.10	-	673.86	649.74	-	610.90
our method	465.15	3220.69	233.80	1684.17	230.89	112.87	799.91	63.82

Table 4.5 Running time results in seconds for average-linkage clustering. Columns indicate data set and input size considered. Missing entries indicate a failure due to a memory error on this input size. All other baseline algorithms fail on these large sizes.

Near cluster search. To test our near cluster search algorithm we considered the following setup. First, for each dataset we constructed a clustering using a subsample of 50k points[5] to get a collection of clusters and a query, we ran k-means ($k = 11$) and took one of the clusters to be the query cluster Q and the remaining k clusters to be the clustering C. For our algorithm we measured running time of constructing the embedding plus answering the query. This is compared with the running time of the baseline method which answers a query by computing the average distance of Q to each cluster. We compared the quality of the cluster returned by

[5]The Shuttle dataset was used in its entirety as it has less than 50k points

Data Set	Average	95-percentile	Max
Shuttle	1.079	1.268	1.894
Blog	1.070	1.238	2.515
Covertype	1.081	1.330	3.535
Susy (L)	1.143	1.481	2.698
Susy (S)	1.132	1.444	2.465
Higgs (L)	1.194	1.652	8.260
Higgs (S)	1.188	1.632	9.859

Table 4.6 Embedding ratio stats, with Dev( ) approximated with reservoir sampling, at sample size 16384

Data Sizes	64	128	256	512	1024	2048	4096	8192	16384	32768	43500
Shuttle	99.79	99.61	99.78	99.67	99.63	99.76	99.81	99.79	99.62	99.67	99.79
Blog	92.74	95.72	95.84	96.96	97.88	98.32	99.19	99.28	99.59	99.74	/
Covertype	89.67	89.95	92.58	94.52	95.56	96.72	97.65	98.63	99.48	99.57	/
Susy (L)	98.98	98.92	99.26	99.11	98.46	98.47	98.56	98.56	98.32	98.08	/
Susy (S)	98.81	99.23	99.26	98.80	99.00	98.65	98.69	98.41	98.56	98.41	/
Higgs (L)	98.61	98.36	98.32	97.67	97.69	97.78	97.65	97.75	97.21	97.37	/
Higgs (S)	98.21	97.99	98.18	98.04	98.00	97.91	97.57	97.56	97.73	97.48	/

Table 4.7 Mean objective approximation ratio of average-linkage, in percentage

our algorithm to the cluster returned by the baseline algorithm. Finally, we compared the embedded distance of the returned cluster to the true average distance. The parameter k was ranged in powers of 2 from 128 to 4096, and 10 trials were done for each value of k.

The running times of cluster search are in Table 4.1. Our algorithm utilizing the embedding outperforms the baseline method in both average run time and variance. The run time of the naive method varies drastically across different instances, being quadratically dependent on the size of the query cluster and the number of points. Our method has very stable run times, consistent with the theoretical justi cation, as the run time only depends linearly on the size of the query cluster and the number of given clusters.

We display the accuracy of our near cluster search algorithm in Table 4.2. Approx. ratio refers to the ratio of the average-linkage distance of the returned cluster for each query to that of the actual closest cluster. Embedding ratio is the ratio, for the returned cluster, of its distance from the query cluster in the embedding compared to their actual average distance. We report the *worst* such ratio encountered in all instances we tested. Our method has an error rate of about 2% for all data sets, and we observe less than 10% difference in the embedded distances. Near cluster search with the embedding yields very accurate results, much better than the theoretical bounds.

Average-linkage hierarchical clustering. Running time wise, our average-linkage algorithm is faster than all the baseline algorithms on large datasets. See Tables 4.3 and 4.5 for

details. All baseline algorithms also begin to fail once sizes reach 32k to 64k points due to memory requirements. Recall that they require $\Omega(n^2)$ memory for memoization, whereas we need slightly super-linear memory.

Figure 4.1 compares the growth rate of running time versus input size for our implementation and the three baselines. Both running time and input size are plotted on a logarithmic scale. Notice that any polynomial function $y = cx^r$ becomes a linear function on the log-log plot, thus the *slope* of a curve equals the exponent r. Our algorithm has a slope that is close to 1, showing it to have near linear running time. The three baselines share a similar growth rate. The curve "FastCluster Regression Model" is a linear regression fitted to the running times of FastCluster, the fastest of the baselines. We report the fitted slope to be 2, near showing all three baselines to have quadratic running time. Figure 4.1 focused on Susy (L) but the results are similar for all datasets.

Fig. 4.1 Comparison of how running time grows with input size, both axis in log scale, on data set Susy(L)

The embedding gives very accurate estimates of the average distance between clusters. See Table 4.8 for detailed statistics on the embedding ratio. Further, Table 4.6 shows negligible loss in using sampling to approximate the deviation term in the cluster embedding (4.1).

Closeness ratioto test accuracy, in each step of the algorithm, we measure how well the merge our algorithm makes approximate the current minimum average-linkage. The approximation ratio is the ratio of the average distance of the chosen pair of clusters to the minimum average distance. To measure this, we find the minimum average distance d^* only at the start of each round of LSH. Then, for each merge of A and B in this round we compute $\text{Avg}(A,B)/d^*$, giving an *upper bound* on the approximation ratio, named the closeness ratio.

Dataset	Mean	95th percentile	Max
Shuttle	1.008	1.031	1.090
Blog	1.008	1.032	1.123
Covertype	1.008	1.032	1.101
Susy (L)	1.005	1.019	1.088
Susy (S)	1.006	1.020	1.075
Higgs (L)	1.003	1.012	1.058
Higgs (S)	1.003	1.012	1.077

Table 4.8 Embedding ratio stats using true Dev( ) term at size 16384

Dataset	Mean	95th percentile	Max
Shuttle	1.13	1.33	1.58
Blog	1.30	1.85	4.12
Covertype	1.56	2.19	3.65
Susy (L)	1.28	1.61	2.02
Susy (S)	1.25	1.52	1.87
Higgs (L)	1.33	1.65	2.01
Higgs (S)	1.34	1.66	1.99

Table 4.9 Closeness ratio stats, size 1024

Table 4.9 includes more detailed statistics on all subsamples of size 1024. The merges picked by our algorithm are close to the minimum.

Finally, see Table 4.7. We observe a negligible loss in overall tree quality compared to vanilla average-linkage.

Comparing with aother ANN-based baseline. We compare our implementation and the standard implementation of average-linkage on the objective in [59]. Recall that the algorithm has strong theoretical guarantees, has similar designs to our algorithm but utilizes the original ANN schema more. We show that our implementation is way more ef cient.

Input Size	64	128	256	512	1024	2048	4096	8192
Shuttle	41.45	102.75	229.83	634.35	2897.20	7404.18	18432.55	46840.01
Blog	2140.38	6030.19	16089.31	52891.13	-	-	-	-
Covertype	407.44	1204.50	3369.60	13730.49	47937.79	161861.26	-	-
Higgs (S)	37.77	80.92	189.81	511.43	2936.55	8001.01	21468.35	-
Susy (S)	6.48	13.84	31.85	95.76	374.26	977.79	2896.95	8565.05

Table 4.10 Average running time in seconds for algorithm due to [3]. A blank entry indicates that the algorithm crashed due to running out of memory at this input size.

We give our results for the algorithm in [3] in Table 4.10. We present the average running time across ve sampled instances at each input size. Note that these times are signi cantly worse than our algorithm as shown in Section 4.7. Moreover, this algorithm was unable to scale to large input sizes over 100k points as our algorithm did. Also observe that the performance of this method signi cantly degrades on higher dimensional data sets such as Blog and Covertype, which is due to the signi cant increase in the dimension the algorithm carries out.

4.8 Conclusions and Future Work

This paper introduces a sparse cluster embedding that approximately preserves the average distance between clusters. The embedding can be computed in linear time and, after the embedding, pairwise average cluster distances can be computed in $O(d)$ time.

The embedding enables a sub-quadratic average-linkage algorithm. In experiments, the algorithm scales better than current popular implementations of average-linkage.

An interesting direction for future work is to explore cluster embeddings further and determine whether they can be leveraged in other data mining tasks.

Chapter 5

Relational Algorithms For K-Means Clustering

5.1 Introduction

Kaggle surveys[2] show that the majority of learning tasks faced by data scientists involve relational data. Conventional formats usually represent data with multi-dimensional points where each dimension corresponds to a feature of the data. In contrast, a relational database consists of tables $T_1, T_2, \ldots, T_m$ where the features could be stored partially in the tables. The columns in each table are a subset of features[1] and the rows are data records for these features. The underlying data is represented by the design matrix $J = T_1 \bowtie \cdots \bowtie T_m$ where each row in J can be interpreted as a data point. Here join ($\bowtie$) is a binary operator on two tables T_i and T_j. The result of the join is the set of all possible concatenations of two rows from T_i and T_j such that they are equal in their common columns/features. If T_i and T_j have no common columns their join is the cross product of all rows. See Table 5.1 for an example of join operation on two tables.

Almost all learning tasks are designed for data in matrix format. The current standard practice for a data scientist is the following.

Standard Practice:

1. Extract the data points from the relational database by taking the join of all tables to find the design matrix $J = T_1 \bowtie \cdots \bowtie T_m$.

[1] In relational database context the columns are also referred as attributes but here we call them features per the tradition of broader communities.

T_1	
f_1	f_2
1	1
2	1
3	2
4	3
5	4

T_2	
f_2	f_3
1	1
1	2
2	3
5	4
5	5

$T_1 \bowtie T_2$		
f_1	f_2	f_3
1	1	1
1	1	2
2	1	1
2	1	2
3	2	3

Table 5.1 A join of tables T_1 and T_2. Each has 5 rows and 2 features, sharing f_2. The join has all features from both tables. The rows with $f_2 = x$ in the join is the cross product of all rows with $f_2 = x$ from T_1 and T_2. For example, for $f_2 = 1$, the four rows in $T_1 \bowtie T_2$ has (f_1, f_3) values $\{(1,1),(1,2),(2,1),(2,2)\}$, this is the cross product of $f_1 \in \{1,2\}$ from T_1 and $f_3 \in \{1,2\}$ from T_2.

2. Then interpret each row of J as a point in a Euclidean space and the columns as the dimensions, corresponding to the features of data.

3. Import this design matrix J into a standard algorithm.

A relational database is a highly compact data representation format. The size of J can be exponentially larger than the input size of the relational database. Explicitly extracting J makes the standard practice inefficient. Theoretically, there is a potential for exponential speed-up by running algorithms directly on the tables in relational data. We call such algorithms relational algorithms if their running time is polynomial in the size of tables when the database is acyclic. Acyclic databases will be defined shortly. This leads to the following exciting algorithmic question.

The Relational Algorithm Question:

A. Which standard algorithms can be implemented as relational algorithms?

B. For standard algorithms that are not implementable by relational algorithms, is there an alternative efficient relational algorithm that has similar performance?

This question has recently been of interest to the community. However, few algorithmic techniques are known. Moreover, we do not have a good understanding of which problems can be solved on relational data and which cannot. Relational algorithm design has a interesting combinatorial structure that requires a deeper understanding.

We design a relational algorithm for k-means. It has a polynomial time complexity for acyclic relational databases. The relational database is acyclic if there exists a tree with the following properties. There is exactly one node in the tree for each table. Moreover, for any feature (i.e. column) f, let $V(f)$ be the set of nodes whose corresponding tables contain

feature f. The subgraph induced on $V(f)$ must be a connected component. Acyclicity can be easily checked, as the tree can be found in polynomial time if it exists [136].

Luckily, most of the natural database schema are acyclic or nearly acyclic. Answering seemingly simple questions on general (cyclic) databases, such as if the join is empty or not is NP-Hard. For general databases, efficiency is measured in terms of the fractional hypertree width of the database (denoted by "fhtw").[2] This measures how close the database structure is to being acyclic. This parameter is 1 for acyclic databases and larger as the database is farther from being acyclic.

State-of-the-art algorithms for queries as simple as counting the number of rows in the design matrix have linear dependency on n^{fhtw} where n is the maximum number of rows in all input tables [8]. Running in time linear in n^{fhtw} is the goal, as fundamental barriers need to be broken to be faster. Notice that this is polynomial time when fhtw is a fixed constant (i.e. nearly acyclic). Our algorithm has linear dependency on n^{fhtw}, matching the state-of-the-art.

Relational algorithm for k-means. k-means is perhaps the most widely used data mining algorithm (e.g. k-means is one of the few models in Google's BigQuery ML package [99]). The input to the k-means problem consists of a collection S of points in a Euclidean space and a positive integer k. A feasible output is k points $c_1, \ldots, c_k$, which we call centers. The objective is to choose the centers to minimize the aggregate squared distance from each original point to its nearest center.

Recall extracting all data points could take time exponential in the size of a relational database. Thus, the problem is to find the cluster centers without fully realizing all of the data points the relational data represents.

[60] was the first chapter to give a non-trivial k-means algorithm that works on relational inputs. The chapter gives a $O(1)$-approximation. The algorithm's running time has super-linear dependency on k^d when the tables are acyclic and thus is not polynomial. Here k is the number of cluster centers and d is the dimension (a.k.a number of features) of the points. This is equivalently the number of distinct columns in the relational database. For a small number of dimensions, this algorithm is a large improvement over the standard practice and they showed the algorithm gives up to 350x speed up on real data versus performing the query to extract the data points (not even including the time to cluster the output points).

Several questions remain. Is there a relational algorithm for k-means? What algorithmic techniques can we use as building blocks to design relational algorithms? Moreover, how can we show some problems are hard to solve using a relational algorithm?

[2] See Appendix A.2 for a formal definition.

Results. The main result of the chapter is the following.

Theorem 5.1.1. Given an acyclic relational database with tables $T_1, T_2, \ldots, T_m$ where the design matrix J has N rows and d columns. Let n be the maximum number of rows in any table. Then there is a randomized algorithm running in time polynomial in d, n and k that computes an $O(1)$ approximate k-means clustering solution with high probability.

In Appendix A.2, we discuss the algorithm's time complexity for cyclic databases.

Challenges. To illustrate the challenges for finding such an algorithm as described in the prior theorem, even when the database is acyclic, consider the following theorem.

Theorem 5.1.2. Given an acyclic relational database with tables $T_1, T_2, \ldots, T_m$ where the design matrix J has N rows and d columns. Given k centers $c_1, \ldots, c_k$, let J_i be the set of points in J that are closest to c_i for $i \in [k]$. It is #P-Hard to compute $|J_i|$ for $k \geq 2$ and NP-Hard to approximate $|J_i|$ to any factor for $k \geq 3$.

The full proof is postponed until Section 5.2.1. We show this by reducing an NP-hard problem to the problem of determining if J_i is empty or not. Counting points closest to a center is a fundamental building block in almost all k-means algorithms. Moreover, we show even performing one iteration of the classic Lloyd's algorithm is #P-hard.

Theorem 5.1.3. Given an acyclic join, and two centers, it is #P-hard to compute the center of mass for the points assigned to each center.

Together these two theorems necessitates the design of new techniques to address the main theorem, shows that seemingly trivial algorithms are difficult relationally, and suggests computing a coreset is the right approach for the problem as it is difficult to cluster the data directly.

Overview of techniques. We first compute a coreset of all points in J. That is, a collection of points with weights such that if we run an $O(1)$ approximation algorithm on this weighted set, we will get a $O(1)$ approximate solution for all of J. To do so, we sample points according to the principle in k-means++ algorithm and assign weights to the points sampled. The number of points chosen will be $O(k \log N)$. Any $O(1)$-approximate weighted k-means algorithm can be used on the coreset to give Theorem 5.1.1.

k-means++: k-means++ is a well-known k-means algorithm [22, 10]. The algorithm iteratively chooses centers $c_1, c_2, \ldots$. The first center c_1 is picked uniformly from J. Given

that $c_1, \ldots, c_{i-1}$ are picked, a point x is picked as c_i with probability $P(x) = \frac{L(x)}{Y}$ where $L(x) = \min_{j \in [i-1]} (\|x - c_j\|_2^2)$ and $Y = \sum_{x \in J} L(x)$. Here $[i-1]$ denotes $\{1, 2, \ldots, i-1\}$.

Say we sample $Q(k \log N)$ centers according to this distribution, which we call the k-means++ distribution. It was shown in [10] that if we cluster the points by assigning them to their closest centers, the total squared distance between points and their cluster centers is at most $O(1)$ times the optimal k-means cost with high probability. Note that this is not a feasible k-means solution because more than k centers are used. However, leveraging this, the work showed that we can construct a coreset by weighting these centers according to the number of points in their corresponding clusters.

We seek to mimic this approach with a relational algorithm. Let's focus on one iteration where we want to sample the center given $c_1, \ldots, c_{i-1}$ according to the k-means++ distribution. Consider the assignment of every point to its closest center in $c_1, \ldots, c_{i-1}$. Notice that the k-means++ probability is determined by this assignment. Indeed, the probability of a point being sampled is the cost of assigning this point to its closest center $(\min_{j \in [i-1]} \|x - c_j\|_2^2)$ normalized by Y. Y is the summation of this cost over all points.

The relational format makes this distribution difficult to compute without the design matrix J. It is hard to efficiently characterize which points are closest to which centers. The assignment partitions the data points according to their closest centers, where each partition may not be easily represented by a compact relational database (unlike J).

A Relational k-means++ Implementation: Our approach will sample every point according to the k-means++ distribution without computing this distribution directly. Instead, we use rejection sampling [39], which allows one to sample from a "hard" distribution P using an "easy" distribution Q. Rejection sampling works by sampling from Q first, then reject the sample with another probability used to bridge the gap between Q and P. The process is repeated until a sample is accepted. In our setting, P is the k-means++ distribution, and we need to find a Q which could be sampled from efficiently with a relational algorithm (without computing J). Rejection sampling theory shows that for the sampling to be efficient, Q should be close to P point-wise to avoid high rejection frequency. In the end, we will perfectly simulate the k-means++ algorithm.

We now describe the intuition for designing such a Q. Recall that P is determined by the assignment of points to their closest centers. We will approximate this assignment up to a factor of $O(i^2 d)$ when sampling the i^{th} center c_i, where d is the number of columns in J. Intuitively, the approximate assignment makes things easier since for any center we can easily find the points assigned to it using an efficient relational algorithm. The Q is found by normalizing the squared distance between each point and its assigned center.

The approximate assignment is designed as follows. Consider the d-dimensional Euclidean space where the data points in J are located. The algorithm divides space into a laminar collection of hyper-rectangles[3] (i.e., $\{x \in \mathbb{R}^d : v_j \leq x_j \leq w_j, j = 1, \ldots, d\}$, here x_j is the value for feature f_j). We assign each hyper-rectangle to a center. A point assigns itself to the center that corresponds to the smallest hyper-rectangle containing the point.

The key property of hyper-rectangles that benefits our relational algorithm is: we can efficiently represent all points from J inside any hyper-rectangle by removing some entries in each table from the original database and taking the join of all tables. For example, if a hyper-rectangle has constraint $v_j \leq x_j \leq w_j$, we just remove all the rows with value outside of range $[v_j, w_j]$ for column f_j from the tables containing column f_j. The set of points assigned to a given center can be found by adding and subtracting a laminar set of hyper-rectangles, where each hyper-rectangle can be represented by a relational database.

Weighting the Centers: We have sampled a good set of cluster centers. In order to get a coreset we need to assign weights to them. As we have already mentioned, assigning the weights cannot be computed relationally. In fact, they cannot be approximated up to any factor in polynomial time unless $P = NP$. Rather, we design an alternative relational algorithm for computing the weights. Each weight will not be an approximate individually, but we prove that the weighted centers form a $(1+\varepsilon)$-approximate coreset in aggregate.

The main algorithmic idea is that for each center c_i we generate a collection of hyperspheres around c_i containing geometrically increasing numbers of points. The space is then partitioned using these hyperspheres where each partition contains a portion of points in J. Using the algorithm from [4], we then sample a poly-log sized collection of points from each partition, and use this subsample to estimate the fraction of the points in this partition which are closer to c_i than any other center. The estimated weight w_i is aggregated accordingly.

Chapter Organization: As relational algorithms are relatively new, we begin with some special cases which help the reader build intuition. In Section 5.2 we give a warm-up by showing how to implement 1-means++ and 2-means++ (i.e. initialization steps of k-means++). In this section, we also prove Theorem 5.1.2 as an example of the limits of relational algorithms. In Section 5.3 we go over background on relational algorithms that our overall algorithm will leverage. In Section 5.4 we give the k-means++ algorithm via rejection sampling. Section 5.5 shows an algorithm to construct the weights and then analyze this algorithm. Many of the technical proofs appear in the appendix due to space.

[3] A laminar set of hyper-rectangles means any two hyper-rectangles from the set either have no intersection, or one of them contains the other.

5.2 Warm-up: Ef ciently Implementing 1-means++ and 2-means++

This section is a warm-up to understand the combinatorial structure of relational data. We will show how to do k-means++ for $k \in \{1, 2\}$ (referred to as 1- and 2-means++) on a simple join structure. We will also show the proof of Theorem 5.1.2 which states that counting the number of points in a cluster is a hard problem on relational data.

First, let us consider relationally implementing 1-means++ and 2-means++. For better illustration, we consider a special type of acyclic table structure named *path join*. The relational algorithm used will be generalized to work on more general join structures when we move to the full algorithm in Section 5.4.

In a path join each table T_i has two features/columns f_i and f_{i+1}. Table T_i and T_{i+1} then share a common column f_{i+1}. Assume for simplicity that each table T_i contains n rows. The design matrix $J = T_1 \bowtie T_2 \bowtie \cdots \bowtie T_m$ has $d = m + 1$ features, one for each feature (i.e. column) in the tables.

Even with this simple structure, the size of the design matrix J could still be exponential in the size of database J could contain up to $n^{m/2}$ rows , and $dn^{m/2}$ entries. Thus the standard practice could require time and space $\Omega(dn^{m/2})$ in the worst case.

T_1	
f_1	f_2
1	1
2	1
3	2
4	3
5	4

T_2	
f_2	f_3
1	1
1	2
2	3
5	4
5	5

$J = T_1 \bowtie T_2$		
f_1	f_2	f_3
1	1	1
1	1	2
2	1	1
2	1	2
3	2	3

Table 5.2 A path join instance where the two tables T_1 and T_2 have $m = 2$ and $n = 5$. This shows T_1, T_2, the design matrix J, and the resulting layered directed graph G. Every path from the left most layer to the right most layer of this graph G corresponds to one data point for the clustering problem (i.e. a row of the design matrix).

Graph Illustration of the Design Matrix: Conceptually consider a directed acyclic graph G, where there is one layer of nodes corresponding to each feature f_i ($i = 1, \ldots, d$), and edges only point from nodes in layer f_i to layer f_{i+1}.

The nodes in G correspond to feature values, and edges in G correspond to rows in tables. There is one vertex v in layer f_i for each value that appears in column f_i in table T_{i-1} or T_i, and one edge pointing from u in layer f_i to v in layer f_{i+1}, if (u, v) is a row in table T_i. Then,

there is a one-to-one correspondence between full paths in G (paths from layer f_1 to layer f_d) and rows in the design matrix.

A Relational Implementation of 1-means++: Implementing the 1-means++ algorithm is equivalent to generating a full path uniformly at random from G. We generate this path by iteratively picking a row from table $T_1, \ldots, T_m$, corresponding to picking an arc pointing from layer f_1 to f_2, f_2 to f_3, ..., such that concatenating all picked rows (arcs) will give a point in J (full path in G).

To sample a row from T_1, for every row $r \in T_1$, consider $r \bowtie J$, which is all rows in J whose values in column (f_1, f_2) are equivalent to r. Let the function $F_1(r)$ denote the total number of rows in $r \bowtie J$. This is also the number of full paths passing arc r. Then, every $r \in T_1$ is sampled with probability $\frac{F_1(r)}{\sum_{r' \in T_1} F_1(r')}$, notice $\sum_{r' \in T_1} F_1(r')$ is the total number of full paths. Let the picked row be r_1.

After sampling r_1, we can conceptually throw away all other rows in T_1 and focus only on the rows in J that uses r_1 to concatenate with rows from other tables (i.e., $r_1 \bowtie J$). For any row $r \in T_2$, let the function $F_2(r)$ denote the number of rows in $r_1 \bowtie r \bowtie J$, also equivalent to the total number of full paths passing arc r_1 and r. We sample every r with probability $\frac{F_2(r)}{\sum_{r' \in T_2} F_2(r')}$. Notice that $\sum_{r' \in T_2} F_2(r') = F_1(r_1)$, the number of full paths passing arc r_1. Repeat this procedure until we have sampled a row in the last table T_m. For table T_i and $r \in T_i$, assuming we have sampled $r_1, \ldots, r_{i-1}$ from $T_1, \ldots, T_{i-1}$ respectively, throw away all the other rows in previous tables and focus on $r_1 \bowtie \cdots \bowtie r_{i-1} \bowtie J$. $F_i(r)$ is the number of rows in $r_1 \bowtie \cdots \bowtie r_{i-1} \bowtie r \bowtie J$ and r is sampled with probability proportional to $F_i(r)$. It is easy to verify that every full path is sampled uniformly.

For every table T_i we need to find the function $F_i(\cdot)$ which is defined on all its rows. There are m such functions. For each $F_i(\cdot)$, we can find all $F_i(r)$ values for $r \in T_i$ using a one-pass dynamic programming and then sample according to the values. Repeating this procedure k rounds completes the sampling process. This gives a polynomial time algorithm.

A Relational Implementation for 2-means++: Assume $x = (x_1, \ldots, x_d)$ is the first center sampled and now we want to sample the second center. By 2-means++ principles, any row $r \in J$ is sampled with probability $\frac{\|r - x\|^2}{\sum_{r' \in J} \|r' - x\|^2}$. For a full path in G corresponding to a row $r \in J$ we refer to $\|r - x\|^2$ as the aggregated cost over all d nodes/features.

Similar to 1-means++, we pick one row in each table from T_1 to T_m and putting all the rows together gives us the sampled point. Assume we have sampled the rows $r_1, r_2, \ldots, r_{i-1}$ from the first $i-1$ tables and we focus on all full paths passing $r_1 \bowtie \cdots \bowtie r_{i-1}$ (i.e., the new design matrix $r_1 \bowtie \cdots \bowtie r_{i-1} \bowtie J$). In 1-means++, we compute $F_i(r)$ which is the total number of full paths passing arc $r_1, \ldots, r_{i-1}, r$ (i.e., $r_1 \bowtie \cdots \bowtie r_{i-1} \bowtie r \bowtie J$.) and sample $r \in T_i$ from a distribution normalized using $F_i(r)$ values. In 2-means++, we define $F_i(r)$ to

be the summation of aggregated costs over all full paths which pass $s_1, arcs, r_{i-1}; r$. We sample $r \in T_i$ from a distribution normalized using $F(r)$ values.

It is easy to verify the correctness. Again, each $F(r)$ could be computed using a one-pass dynamic programming which gives the values for all rows in T_i when we sample from T_i. This would involve m rounds of such computations and give a polynomial algorithm.

5.2.1 Hardness of Relationally Computing the Weights

This section established the difficulty of relationally implementing k-means algorithms by proving Theorem 5.1.2 and 5.1.3.

Proof of [Theorem 5.1.2] Here we prove Theorem 5.1.2. We will focus on showing that given a set of centers, counting the number of points in J that is closest to any of them is #P-hard. We prove #P-Hardness by a reduction from the well known #P-hard Knapsack Counting problem. The input to the Knapsack Counting problem consists of a set $W = \{w_1, \ldots, w_h\}$ of nonnegative integer weights, and a nonnegative integer L. The output is the number of subsets of W with aggregate weight at most L. To construct the relational instance, for each $i \in [h]$, we define the tables T_{2i-1} and T_{2i} as follows:

T_{2i-1}	
f_{2i-1}	f_{2i}
0	0
0	w_i

T_{2i}	
f_{2i}	f_{2i+1}
0	0
w_i	0

Table 5.3 NP-hardness Reduction

Let centers c_1 and c_2 be arbitrary points such that points closer to c_1 than c_2 are those points p for which $\sum_{i=1}^{d} p_i \leq L$. Then there are 2^h rows in J, since w_i can either be selected or not selected in feature $2i$. The weight of c_1 is the number of points in J closer to c_1 than c_2, which is in turn exactly the number of subsets of W with total weight at most L.

Then we prove the second part of Theorem 5.1.2 that given an acyclic database and a set of centers $c_1, \ldots, c_k$, it is NP-Hard to approximate the number of points assigned to each center when $k \geq 3$. We prove it by reduction from Subset Sum. In Subset Sum problem, the input is a set of integers $A = \{w_1, \ldots, w_m\}$ and an integer L, the output is true if there is a subset of A such that its summation is L. We create the following acyclic schema. There are m tables. Each table T_i has a single unique column x_i with two rows $w_i, 0$. Then the join of the tables has 2^m rows, and it is a cross product of the rows in different tables in which each row represents one subset of A.

Then consider the following three centers: $c_1 := (\frac{L-1}{m}, \frac{L-1}{m}, \ldots, \frac{L-1}{m})$, $c_2 = (\frac{L}{m}, \ldots, \frac{L}{m})$, and $c_1 = (\frac{L+1}{m}, \frac{L+1}{m}, \ldots, \frac{L+1}{m})$. The Voronoi diagram that separates the points assigned to each of these centers consists of two parallel hyperplanes $\sum_i x_i = L - 1/2$ and $\sum_i x_i = L + 1/2$ where the points between the two hyperplanes are the points assigned to c_2. Since all the points in the design matrix have integer coordinates, the only points that are between these two hyperplanes are those points for which $\sum_i x_i = L$. Therefore, the approximation for the number of points assigned to c_2 is non-zero if and only if the answer to Subset Sum is True. □

Proof of [Theorem 5.1.3] We prove by a reduction from a decision version of the counting knapsack problem. The input to the counting knapsack problem consists of a set $W = \{w_1, \ldots, w_n\}$ of positive integer weights, a knapsack size L, and a count D. The problem is to determine whether there are at least D subsets of W with aggregate weight at most L. The points in our instance of k-means will be given relationally. We construct a join query with $n+1$ columns/attributes, and n tables. All the tables have one column in common and one distinct column. The i-th table has 2 columns (d_i, d_{n+1}) and three rows $\{(w_i, -1), (0, -1), (0, D)\}$. Note that the join has 2^n rows with -1 in dimension $n+1$, and one row with values $(0, 0, \ldots, 0, D)$. The rows with -1 in dimension $d+1$ have all the subsets of $\{w_1, \ldots, w_n\}$ in their first n dimensions. Let the two centers for k-means problem be any two centers c_1 and c_2 such that a point x is closer to c_1 if it satisfies $\sum_{d=1}^{n} x_d < L$ and closer to c_2 if it satisfies $\sum_{d=1}^{n} x_d > L$. Note that the row $(0, 0, \ldots, 0, D)$ is closer to c_1. Therefore, the value of dimension $n+1$ of the center of mass for the tuples that are closer to c_1 is $Y = (D - C)/C$ where C is the actual number of subsets of W with aggregate weight at most L. If Y is negative, then the number of solutions to the counting knapsack instance is at least D. □

5.3 Related Work and Background

Related Work on K-means: Constant approximations are known for the k-means problem in the standard computational setting [102, 84]. Although the most commonly used algorithm in practice is a local search algorithm called Lloyd's algorithm, or sometimes confusingly just called "the k-means algorithm". The k-means++ algorithm from [22] is a $O(\log k)$ approximation algorithm, and is commonly used in practice to seed Lloyd's algorithm. Some coreset construction methods have been used before to design algorithms for the k-means problem in other restricted access computational models, including streaming [72, 36], and the MPC model [66, 26], as well as speeding up sequential methods [111, 128].

Relational Algorithms for Learning Problem: Training different machine learning models on relational data has been studied; however, many of the proposed algorithms are not efficient under our definition of a relational algorithm. It has been shown that using repeated patterns in the design matrix, linear regression, and factorization machines can be implemented [121] more efficiently. [92, 126, 6] has improved the relational linear regression and factorization machines for different scenarios. A unified relational algorithm for problems such as linear regression, singular value decomposition and factorization machines proposed in [7]. Algorithms for training support vector machine is studied Yang et al. [5]. In [50], a relational algorithm is introduced for Independent Gaussian Mixture Models, and they have shown experimentally that this method will be faster than materializing the design matrix.

Relational Algorithm Building Blocks: In the path join scenario, the 1-means++ and 2-means++ sampling methods introduced in subsection 5.2 have similar procedures: starting with the first table T_1, iteratively evaluate some general function $F_i(\cdot)$ defined on all rows in the table T_i, sample one row w_i according to the distribution normalized from $F_i(\cdot)$. The function $F_i(\cdot)$ for table T_i is defined on the matrix $x_1 \bowtie \cdots \bowtie r_{i-1} \bowtie J$ where J is the design matrix. This matrix is also the design matrix of a new relational database, constructed by throwing away all rows in previous tables apart from the sampled $r_1, \ldots, r_{i-1}$.

We can generalize the computation $F_i(\cdot)$ functions into a broader class of queries that we know could be implemented efficiently on any acyclic relational databases, namely SumProd queries See [8] for more details. In the following lemmas assume the relational database has tables $T_1, \ldots, T_m$ and their design matrix is J, let n be the maximum number of rows in each table T_i, m be the number of tables and d be the number of columns in J.

Definition 5.3.1. For the j^{th} feature ($j \in [d]$) let $q_j : \mathbb{R} \to S$ be an efficiently computable function that maps feature values to some set S. Let the binary operations $\oplus$ and $\otimes$ be any operators such that $(S, \oplus, \otimes)$ forms a commutative semiring. The value of $\bigoplus_{x \in J} \bigotimes_{j \in [d]} q_j(x_j)$ is a SumProd query.

Lemma 5.3.1([8]). Any SumProd query can be computed efficiently in time $O(md^2 n^{fhtw} \log(n))$ where fhtw is the fractional hypertree width of the database. For acyclic databases fhtw=1 so the running time is polynomial.

Despite the cumbersome formal definition of SumProd queries, below we list their key applications used in this paper. With a little abuse of notation, throughout this paper we use $Y(n, d, m)$ to denote the worst-case time bound on any SumProd queries.

Lemma 5.3.2. Given a point $y \in \mathbb{R}^d$ and a hyper-rectangle $H = \{x \in \mathbb{R}^d : v_i \le x_i \le w_i, i = 1, \ldots, d\}$ where v and w are constant vectors, we let $\mathcal{H}$ denote the data points represented

by rows of J that also fall into b. Pick any table T_j. Using one single SumProd query we can compute for all $t \in T_j$ the value $\sum_{p \in m \ltimes J \ltimes b} \|p - y\|_2^2$. The time required is at most that required by one SumProd query $Y(n,d,m)$.

Lemma 5.3.2 is an immediate result of Theorem A.2.3 and the fact that we can efficiently represent all points from J inside any hyper-rectangle by removing some entries in each table from the original database and taking the join of all tables. The following lemma follows by an application of the main result in [4]. In Appendix A.1 we formally show to apply their result to give the following lemma.

Lemma 5.3.3 ([4]). Given a hypersphere $\{x \in \mathbb{R}^d : \|x - y_0\|^2 = z_0^2\}$ where y_0 is a given point and z_0 is the radius, a $(1+\varepsilon)$-approximation of the number of points in J that lie inside this hypersphere could be computed in $\tilde{O}\left(\frac{m^6 \log^4 n}{\varepsilon^2} Y(n,d,m)\right)$ time.

Notice that a SumProd query could be used to output either a scalar (similar to Lemma 5.3.3) or a vector whose entries are function values for every row in a chosen table T_j (in Lemma 5.3.2). We say the SumProd query is *grouped by* T_j in the latter case.

5.4 Simulating the k-means++ Algorithm

In this section we describe a relational implementation of the k-means++ algorithm. To give some intuition, we first explained the special case of implementing the algorithm when $k = 3$. Then we generalize this to arbitrary value of k.

5.4.1 Relational Implementation of 3-means++

Recall that the 3-means++ algorithm picks a point to be the third center c_3 with probability $P(x) = \frac{L(x)}{Y}$ where $L(x) = \min(\|x - c_1\|_2^2, \|x - c_2\|_2^2)$ and $Y = \sum_{x \in J} L(x)$ is a normalizing constant. Conceptually think of P as being a "hard" distribution to sample from.

Description of the Implementation: The implementation first constructs two identically-sized axis-parallel hypercubes/boxes b_1 and b_2 centered around c_1 and c_2 that are as large as possible subject to the constraints that the side lengths have to be non-negative integral powers of 2, and that b_1 and b_2 can not intersect. Such side lengths could be found since we may assume c_1 and c_2 have integer coordinates or they are sufficiently far away from each other that we can scale them and increase their distance. Conceptually the implementation also considers a box b_3 that is the whole Euclidean space.

Fig. 5.1 Boxes used for sampling the third center

To define our "easy" distribution Q, for each point x define $R(x)$ to be

$$R(x) = \begin{cases} \|x - c_1\|_2^2 & x \in b_1 \\ \|x - c_2\|_2^2 & x \in b_2 \\ \|x - c_1\|_2^2 & x \in b_3 \text{ and } x \notin b_1 \text{ and } x \notin b_2 \end{cases}$$

In the above definition, note that when $x \notin b_1$ and $x \notin b_2$, the distance of x to both centers are relatively similar; therefore, we can assign x to either of the centers – here we have assigned it to c_1. Then $Q(x)$ is defined to be $\frac{R(x)}{Z}$, where $Z = \sum_{x \in J} R(x)$ is normalizing constant. The implementation then repeatedly samples a point x with probability $Q(x)$. After sampling x, the implementation can either (A) reject x and then resample or (B) accept x, which means setting the third center c_3 to be x. The probability that x is accepted after it is sampled is $\frac{L(x)}{R(x)}$, and thus the probability that x is rejected is $1 - \frac{L(x)}{R(x)}$.

It is straightforward to see how to compute b_1 and b_2 (note that b_1 and b_2 can be computed without any relational operations), and how to compute $L(x)$ and $R(x)$ for a particular point x. Thus, the only non-straight-forward part is sampling a point x with probability $Q(x)$, which we explain now:

- The implementation uses a SumProd query to compute the aggregate 2-norm squared distance from c_1 constrained to points in b_3 (all the points) and grouped by table T_1 using Lemma 5.3.2. Let the resulting vector be D. So C_r is the aggregate 2-norm squared distance from c_1 of all rows in the design matrix that are extensions of row r in T_1.

- Then the implementation uses a SumProd query to compute the aggregated 2-norm squared distance from c_2, constrained to points in b_2, and grouped by T_1. Let the

resulting vector be D. Notice that an axis-parallel box constraint can be expressed as a collection of axis-parallel hyperplane constraints, and for every axis-parallel constraint it is easy to remove the points not satisfying it from the join by filtering one of the input tables having that dimension/feature. Then the sum product query is the same as the sum product query in the previous step.

- Then the implementation uses a SumProd query to compute the aggregated 2-norm squared distance from c_1, constrained to points in b_2, and grouped by T_1. Let the resulting vector be E.
- Then pick a row r of T_1 with probability proportional to $C_r \quad E_r + D_r$.
- The implementation then replaces T_1 by a table consisting only of the picked row r.
- The implementation then repeats this process on table T_2, then table T_3 etc.
- At the end, J will consist of one point/row x, where the probability that a particular point x ends up as this final row is $Q(x)$. To see this note that in the iteration performed for T_i, $C \quad E$ is the aggregate 2-norm squared distances from c_1 for all points not in b_2 grouped by T_i, and D is the aggregated squared distances of the points in b_2 to c_2 grouped by T_i.

We now claim that this implementation guarantees that x with probability $P(x)$. We can see this using the standard rejection sampling calculation. At each iteration of sampling from Q, let $S(x)$ be the event that point x is sampled and $A(x)$ be the event that x is accepted. Then,

$$\Pr[S(x) \text{ and } A(x)] = \Pr[A(x) \mid S(x)] \Pr[S(x)] = \frac{L(x)}{R(x)} Q(x) = \frac{L(x)}{Z}$$

Thus x is accepted with probability proportional to $L(x)$, as desired.

As the number of times that the implementation has to sample from Q is geometrically distributed, the expected number of times that it will have to sample is the inverse of the probability of success, which is $\max_x \frac{R(x)}{L(x)}$. It is not too difficult to see (we prove it formally in Lemma 5.4.3) that $\max_x \frac{R(x)}{L(x)} = O(d)$. It takes $3m$ SumProd queries to sample from Q. Therefore, the expected running time of our implementation of 3-means++ is $O(mdY(n;d;m))$.

5.4.2 Simulating k-means++

It is sufficient to explain how center c_i is picked given the previous centers. Recall that the k-means++ algorithm picks a point x to be the center c_i with probability $P(x) = \frac{L(x)}{Y}$ where $L(x) = \min_{j \in [i-1]} (\|x - c_j\|_2^2)$ and $Y = \sum_{x \in J} L(x)$ is a normalizing constant. The implementation consists of two parts. The first part, described in subsubsection The second part, described in subsubsection 5.4.2 samples according to probability distribution P using rejection sampling and an "easy" distribution Q that is derived from the boxes constructed in the first part.

Box Construction

Here we explain the algorithm to construct a set of laminar boxes given the centers sampled in the previous rounds. Note that the centers are explicitly present and we don't need any relational operation for this algorithm.

Algorithm Description: The algorithm maintains two collections G_i and B_i of tuples consisting of a box and a point in that box that we refer to as the representative of the box. When the algorithm terminates B_i will be a laminar collection of boxes that we will use to define the "easy" probability distribution Q.

Initially G_i consists of a hypercube centered at each previous center $c_j, j \in [i-1]$ where each $d-1$ dimensional simplex is at distance 1 from c_j, with the representative point being c_j. And initially B_i is empty. Without loss of generality we can scale so that no pair of these boxes intersect. Over time, some of the boxes in G_i will grow in size, some boxes will be added to G_i and some boxes will be moved from G_i to B_i. So one can think of G_i as a collection of active boxes that might change in the future, and think of B_i as a collection of inactive boxes that are frozen.

The algorithm repeats the following steps. If there are no pair of boxes in G_i that intersect, then a doubling step is performed. In a doubling step every box in G_i is doubled, which means that each $d-1$ dimensional simplex is moved twice as far away from its representative point. Otherwise the algorithm picks two arbitrary intersecting boxes from G_i, say b_1 with representative r_1 and b_2 with representative r_2, and executes what we call a meld step. A meld step consists of

- Computing the smallest box b_3 that contains both b_1 and b_2.
- Adding (b_3, r_1) to G_i.
- Deleting (b_1, r_1) and (b_2, r_2) from G_i.

- If b_1 was created before the last doubling step (that is, b_1 was not melded with another box since the last doubling step), the implementation computes a box b_1' from b_1 by halving, which means that each $d-1$ dimensional simplex is moved so that its distance to the box's representative is halved. Then (b_1', r_1) is added to B_i.
- If b_2 was created before the last doubling step, then the implementation computes a box b_2' from b_2 by halving, which means that each $d-1$ dimensional simplex is moved so that its distance to the box's representative is halved. Then (b_2', r_2) is added to B_i.

The algorithm terminates when there is only one element (b, r) left in G_i, at which point the algorithm adds a box that contains the whole Euclidean space with representative r to B_i. Note that at each iteration of the doubling and melding, the boxes which are added to B_i are the ones that after doubling were melded with other boxes (and they are added at their size before doubling)□

Lemma 5.4.1. The collection of boxes in B_i constructed the above algorithm is laminar.

Proof of [Lemma 5.4.1]

Note that right before each doubling step, the boxes in G_i are disjoint and that is because the algorithm in the previous iteration melds all the boxes that have intersection with each other. Now we prove by induction that for every box b in B_i there exist a box b' in G_i such that $b \subseteq b'$. Since the boxes added to B_i in each iteration are a subset of the boxes in G_i before the doubling step, Laminarity of B_i is a straight-forward consequence.

Now we prove that in all time for every box b in B_i there exist a box b' in G_i such that $b \subseteq b'$. Initially B_i is empty and therefore the claim holds. Assume in some arbitrary iteration ℓ this claim holds right before the doubling step, then after the doubling step since every box in G_i still covers all of the area it was covering before getting doubled, the claim holds. Furthermore, in the melding step every box b_3 that is resulted from melding of two boxes b_1 and b_2 covers both b_1 and b_2; therefore, b_3 will cover b_1 and b_2 if they are added to B_i, and if a box in B_i was covered by either of b_1 or b_2, it will be still covered by b_3. □

Sampling

To define our easy distribution Q, let $b(x)$ be the minimal box in B_i that contains point x and let $r(x)$ the representative of $b(x)$. Define $R(x) = \|x - r(x)\|_2^2$, and $Q(x) = \frac{R(x)}{Z}$ where $Z = \sum_{x \in J} R(x)$ is a normalizing constant.

Implementation Description: The implementation then repeatedly samples a point x with probability $Q(x)$. After sampling x, the implementation can either (A) reject, and then

resample or (B) accept x, which means setting the next center to x. The probability that x is accepted after it is sampled is $\frac{L(x)}{R(x)}$, and thus the probability that x is rejected is $1 - \frac{L(x)}{R(x)}$.

If $S(x)$ is the the event of initially sampling x from distribution Q, and $A(x)$ is the event of subsequently accepting x, we can calculate the probability of accepting x in a particular round using the standard rejection sampling calculation:

$$\Pr[S(x) \text{ and } A(x)] = \Pr[A(x) \mid S(x)] \Pr[S(x)] = \frac{L(x)}{R(x)} Q(x) = \frac{L(x)}{Z}$$

Thus we can see that the probability that x is sampled is proportional to $L(x)$, as desired.

We now explain how to relationally implement the generation of a point x with probability $Q(x)$. The implementation first generates a single row from table T_1, then a single row from table T_2, etc. Then x would be the concatenation of these rows (join of them). So it is sufficient to explain how to implement the generation of a row from an arbitrary table T_ℓ. To generate a row from T_ℓ, the implementation recursively computes a vector C_ℓ that has one entry $C_{\ell,r}$ for each row r of T_ℓ. Initially C_ℓ is the all zeros vector. The recursion starts with the box in B_i that is the whole Euclidean space. Assume that it is currently operating on box b with representative r. First C_ℓ is incremented by the aggregate 2-norm squared distances of points in b from r grouped by T_ℓ, which can be computed by a SumProd query with box constraint b and grouped by T_ℓ. Then let $(b_1, r_1), \ldots, (b_h, r_h)$ be the children of (b, r) in the laminar decomposition B_i. If no such boxes exists, then this is a base case of the recursion, and no further action is taken. Otherwise for each $j \in [h]$, C_ℓ is decremented by a the aggregate 2-norm squared distances of points in b_j from r_j grouped by T_ℓ, which can computed by a SumProd query with box constraint b_j grouped by table T_ℓ. Then the implementation recurses on each (b_j, r_j) for $j \in [h]$. Once C_ℓ is computed, then a row r is selected from T_ℓ with probability proportional to $C_{\ell,r}$, and T_ℓ is replaced by a table with a single row r. □

Lemma 5.4.2. Consider the state of the implementation just before it is going to execute doubling step $j + 1$. Consider an arbitrary box b in G_i at this time, and let $h(b)$ be the number of centers in b at this time. Let a be an arbitrary one of these $h(b)$ centers. Then:

A. The distance from a to any $d - 1$ dimensional simplex of b is at least 2^i.

B. Each side length of b is at most $h(b) 2^{j+1}$.

Proof. The first statement is a direct consequence of the definition of doubling and melding since at any point of time the distance of all the centers in a box is at least 2^i. To prove the second statement, we define the assignment of the centers to the boxes as following.

Consider the centers inside each box right before the doubling step. We call these centers, the centers assigned to and denote the number of them by $h'(b)$. When two boxes b_1 and b_2 are melding into box b_3, we assign their assigned centers to b_3.

We prove each side length of is at most $h'(b)2^{j+1}$ by induction on the number j of executed doubling steps. Since $h'(b) = h(b)$ right before each doubling, this will prove the second statement. The statement is obvious in the base case, The statement also obviously holds by induction after a doubling step is incremented and the side lengths double and the number of assigned boxes don't change. It also holds during every meld step because each side length of the newly created larger box is at most the aggregate maximum side lengths of the smaller boxes that are moved to B_i, and the number of assigned centers in the newly created larger box is the aggregate of the assigned centers in the two smaller boxes that are moved to B_i. Note that since for any box all the assigned centers to are inside at all times, $h'(b)$ is the number of centers inside before the next doubling. □

Lemma 5.4.3. For all points x, $R^2(x) \leq O(i^2 d) \cdot L(x)$.

Proof. Consider an arbitrary point x. Let c_ℓ, $\ell \in [i-1]$, be the center that is closest to x under the 2-norm distance. Assume is minimal such that just before the $(j+1)$-th doubling round, x is contained in a box in G_i. We argue about the state of the algorithm at two times, the time just before doubling round and the time just before doubling round $j+1$. Let be a minimal box in G_i that contains x at time, and let y be the representative for box Notice that we assign x to the representative of the smallest box in B_i that contains it, so x will be assigned to y. Indeed, none of the boxes added into before time contains x by the minimality of j, and when box gets added into B_i (potentially after a few more doubling rounds) it still has the same representative By Lemma 5.4.2 the squared distance from from x to r is at most $(i-1)^2 d 2^{2j+2}$. So it is sufficient to show that the squared distance from x to c_ℓ is $\Omega(2^j)$.

Let b' be the box in G_i that contains c_ℓ at time. Note that x could not have been inside b' at time by the definition of and. Then by Lemma 5.4.2 the distance from to the edge of b' at time is at least 2^{2j-2}, and hence the distance from to x is also at least 2^{2j-2} as x is outside of b'. □

Theorem 5.4.4. The expected time complexity for this implementation of k-means++ is $O(k^3 d \cdot \mathrm{mY}(n;d;m))$.

Proof. When picking center c_i, a point x can be sampled with probability $Q(x)$ in time $O(\mathrm{miY}(n;m;d))$ time. This is because B_i is size $O(i)$, as the laminar decomposition can be thought of as a tree with $i-1$ leaves, the implementation needs to group by each of the

tables. By Lemma 5.4.3, the expected number of times that the implementation will have to sample from Q is $O(i^2 d)$. Summing over $i \in [k]$, we get $O(k^3 d m Y(n,d,m))$ □

5.5 Weighting the Centers

Our algorithm samples a collection C of $k' = Q(k \log N)$ centers using the k-means++ sampling described in the prior section. We give weights to the centers to get a coreset.

Ideally, we would compute the weights in the standard way. That is, w_i would denote the number of points that are closest to point c_i among all centers in C. These pairs of centers and weights (c_i, w_i) are known to form a coreset. Unfortunately, as stated in Theorem 5.1.2, computing such w_i's even approximately is NP-hard. Instead, we will find a different set of weights which still form a coreset and are computable.

Next we describe a relational algorithm to compute a collection W' of weights, one weight $w'_i \in W'$ for each center $c_i \in C$. The proof that the centers with these alternative weights (c_i, w'_i) also form a coreset is postponed until the appendix.

Algorithm for Computing Alternative Weights: Initialize the weight w'_i for each center $c_i \in C$ to zero. In the d-dimensional Euclidean space, for each center $c_i \in C$, we generate a collection of hyperspheres (also named balls) $\{B_{i,j}\}_{j \in [\lg N]}$, where $B_{i,j}$ contains approximately 2^j points from J. The space is then partitioned into $\{B_{i,0}, B_{i,1} \setminus B_{i,0}, B_{i,2} \setminus B_{i,1}, \ldots\}$. For each partition, we will sample a small number of points and use this sample to estimate the number of points in this partition that are closer to c_i than any other centers, and thus aggregating w'_i by adding up the numbers. Fix small constants $\varepsilon, \delta > 0$. The following steps are repeated for $j \in [\lg N]$:

- Let $B_{i,j}$ be a ball of radius $r_{i,j}$ centered at c_i. Find an $r_{i,j}$ such that the number of points in $J \setminus B_{i,j}$ lies in the range $[(1-\delta)2^j, (1+\delta)2^j]$. This is an application of Lemma 5.3.3.
- Let t be a constant that is at least 30. A collection $T_{i,j}$ of $\frac{t}{\varepsilon^2} k'^2 \log^2 N$ "test" points are independently sampled following the same approximately uniform distribution with replacement from every ball $B_{i,j}$. Here an "approximately uniform" distribution means one where every point in $B_{i,j}$ is sampled with a probability $p_{p,i,j} \in [(1-\delta)/|B_{i,j}|, (1+\delta)/|B_{i,j}|]$ on each draw. This can be accomplished efficiently similar to the techniques used in Lemma 5.3.3 from [4]. Further elaboration is given in the Appendix A.1.
- Among all sampled points $T_{i,j}$, find $S_{i,j}$, the set of points that lie in the "donut" $D_{i,j} = B_{i,j} \setminus B_{i,j-1}$. Then the cardinality $s_{i,j} = |S_{i,j}|$ is computed.

- Find $t_{i,j}$, the number of points in $S_{i,j}$ that are closer to c_i than any other center in C.
- Compute the ratio $ratio'_{i,j} = \frac{t_{i,j}}{s_{i,j}}$ (if $s_{i,j} = t_{i,j} = 0$ then $ratio'_{i,j} = 0$).
- If $ratio'_{i,j} \geq \frac{1}{2k'^2 \log N}$ then w'_i is incremented by $ratio'_{i,j} \cdot 2^{j-1}$, else w'_i stays the same.

At first glance the algorithm appears naive. w'_i can be significantly underestimated if in some donuts only a small portion of points are closest to c_i, making the estimation inaccurate based on sampling. However, in Appendix 5.5.1, we prove the following theorem which shows that the alternative weights computed by our algorithm actually form a coreset.

The running time of a naive implementation of this algorithm would be dominated by sampling of the test points. Sampling a single test point can be accomplished with m applications of the algorithm from [4] and setting the approximation error to $\delta = \epsilon/m$. Recall the running time of the algorithm from [4] is $O\left(\frac{m^6 \log^4 n}{d^2} Y(n,d,m)\right)$. Thus, the time to sample all test points is $O\left(\frac{k'^2 m^9 \log^6 n}{\epsilon^4} Y(n,d,m)\right)$. Substituting for k', and noting that $N = n^m$, we obtain a total time for a naive implementation of $O\left(\frac{k^2 m^{11} \log^8 n}{\epsilon^4} Y(n,d,m)\right)$.

5.5.1 Analysis of the Weighting Algorithm

The goal in this subsection is to prove the following Theorem 5.5.1 which states that the alternative weights form an $O(1)$-approximate coreset with high probability. Throughout our analysis, "with high probability" means that for any constant $\alpha > 0$ the probability of the statement not being true can be made less than $\frac{1}{N^\alpha}$ asymptotically by appropriately setting the constants in the algorithm.

Theorem 5.5.1. *The centers C, along with the computed weights W', form an $O(1)$-approximate coreset with high probability.*

Intuitively if a decent fraction of the points in each donut are closer to center c_i than any other center, then Theorem 5.5.1 can be proven by using a straight-forward application of Chernoff bounds to show that each alternate weight w'_i is likely close to the true weight w_i. The conceptual difficultly is if only a very small portion of points in a donut $D_{i,j}$ are closer to c_i than any other points, in which case the estimated ratio $ratio'_{i,j} < \frac{1}{2k'^2 \log N}$ and thus the "uncounted" points in $D_{i,j}$ would contribute no weight to the computed weight w'_i. We call this the *undersampled* case. If many docuts around a center c_i are undersampled, the computed weight w'_i may well poorly approximate the actual weight w_i.

To address this, we need to prove that omitting the weight from these uncounted points does not have a significant impact on the objective value. We break our proof into four parts.

The first part, described in subsubsection 5.5.1, involves conceptually defining a fractional weight w_i^f for each center $c_i \in C$. Each point has a weight of 1, and instead of giving all this weight to its closes center, we allow fractionally assigning the weight to various "near" centers. w_i^f is then the aggregated weight over all points for c_i. The second part, described in subsubsection 5.5.1, establishes various properties of the fractional weight that we will need. The third part, described in subsubsection 5.5.1, shows that each fractional weight w_i^f is likely to be closely approximated the computed weight w_i^0. The fourth part, described in subsubsection 5.5.1, shows that the fractional weights for the centers in C form a $O(1)$-approximate coreset. Subsubsection 5.5.1 also contains the proof of Theorem 5.5.1.

Defining the Fractional Weights

To define the fractional weights we first define an auxiliary directed acyclic graph $G = (S, E)$ where there is one node S corresponding to each row in P. For the rest of this section, with a little abuse of notation we use S to denote both the nodes in graph G, and the set of d-dimensional data points in the design matrix. Let $p \in S \subset C$. Let $a(p)$ denote the subscript of the center closest to p, i.e., if $c_i \in C$ is closest to p then $a(p) = i$. Let $D_{i,j}$ be the donut around c_i that contains p. If $D_{i,j}$ is not undersampled then p will have one outgoing edge (p, c_i). So let us now assume that $D_{i,j}$ is undersampled. Defining the outgoing edges from p in this case is a bit more complicated.

Let $A_{i,j}$ be the points $q \in D_{i,j}$ that are closer to c_i than any other center in C (i.e., $a(q) = i$). If $j = 1$ then $D_{i,1}$ contains only the point p, and the only outgoing edge from p goes to c_i. So let us now assume $j > 1$. Let c_h the center that is closest to the most points in $D_{i,j-1}$, the next donut in toward c_i from $D_{i,j}$. That is $c_h = \arg\max_{c_j \in C} \sum_{q \in D_{i,j-1}} 1_{a(q)=c_j}$. Let $M_{i,j-1}$ be points in $D_{i,j-1}$ that are closer to c_h than any other center. That is $M_{i,j-1}$ is the collection of $q \in D_{i,j-1}$ such that $a(q) = h$. Then there is a directed edge from p to each point in $M_{i,j-1}$. Before defining how to derive the fractional weights from G, let us take a detour to note that G is acyclic.

Lemma 5.5.2. *G is acyclic.*

Proof. Consider a directed edge $(p, q) \in E$, and c_i be the center in C that p is closest to, and $D_{i,j}$ the donut around c_i that contains p. Then since $p \in D_{i,j}$ it must be the case that $\|p - c_i\|_2^2 > r_{i,j-1}$. Since $q \in B_{i,j-1}$ it must be the case that $\|q - c_i\|_2^2 \leq r_{i,j-1}$. Thus $\|p - c_i\|_2^2 > \|q - c_i\|_2^2$. Thus the closest center to q must be closer to q than the closest center to p is to p. Thus as one travels along a directed path in G, although identify of the closest center can change, the distance to the closest center must be monotonically decreasing. Thus, G must be acyclic. □

We explain how to compute a fractional weight w_p^f for each point $p \in S$ using the network G. Initially each w_p^f is set to 1. Then conceptually these weights flow toward the sinks, C, in splitting evenly over all outgoing edges at each vertex. More formally, the following flow step is repeated until is no longer possible to do so:

Flow Step: Let $p \in S$ be an arbitrary point that currently has positive fractional weight and that has positive outdegree h in G. Then for each directed edge (p,q) in G increment w_q^f by w_p^f/h. Finally set w_p^f to zero.

As the sinks in G are exactly the centers in C, the centers in C will be the only points that end up with positive fractional weight. Thus we will use w_c^f to refer to the resulting fractional weight on center $c_i \in C$.

Properties of the Fractional Weights

Let $ratio_{i,j}$ be the fraction of points that are closest to c_i among all centers in C in this donut $D_{i,j} = B_{i,j} - B_{i,j-1}$. We show in Lemma 5.5.3 and Lemma 5.5.5 that with high probability, either the estimated ratio is a good approximation of $ratio_{i,j}$, or the real ratio $ratio_{i,j}$ is very small.

We show in Lemma 5.5.7 that the maximum flow through any node is bounded by $1+\varepsilon$ when N is big enough. This follows using induction because each point $W(\log N)$ neighbors and every point can have in degree from one set of nodes per center. We further know every point that is not uncounted actually contributes to their centers weight.

Lemma 5.5.3. With high probability either $|ratio_{i,j} - ratio'_{i,j}| \le \varepsilon ratio_{i,j}$ or $ratio'_{i,j} \le \frac{1}{2k'^2 \log N}$.

To prove Lemma 5.5.3, we use the following Chernoff Bound.

Lemma 5.5.4. Consider Bernoulli trials $X_1, \ldots, X_n$. Let $X = \sum_{i=1}^{n} X_i$ and $m = E[X]$. Then, for any $\lambda > 0$:

$$\Pr[X \ge m + \lambda] \le \exp\left(\frac{\lambda^2}{2m + \lambda}\right) \quad \text{Upper Chernoff Bound}$$

$$\Pr[X \le m - \lambda] \le \exp\left(\frac{\lambda^2}{3m}\right) \quad \text{Lower Chernoff Bound}$$

Proof. Proof of Lemma 5.5.3: Fix any center $c_i \in C$ and $j \in [\log N]$. By applying the low Chernoff bound from Lemma 5.5.4 it is straight forward to conclude that if t is large then with high probability at least a third of the test points in each $T_{i,j}$ are in the donut $D_{i,j}$. That is, with high probability $s_{i,j} \ge \frac{t}{3\varepsilon^2} k'^2 \log^2 N$. So let us consider a particular $T_{i,j}$ and condition $s_{i,j}$ having some fixed value that is at least $\frac{t}{3\varepsilon^2} k'^2 \log^2 N$. So $s_{i,j}$ is conditioned on being large.

Recall $t_{i,j} = \sum_{p \in W_{i,j}} (1_{p \in T_{i,j}})(1_{a(p)=i})$, and the indicator random variables $1_{p \in T_{i,j}}$ are Bernoulli trials. Further note by the definition of $g_{p,i,j}$ it is the case that $E[t_{i,j}] = \sum_{p \in W_{i,j}} g_{p,i,j}(1_{a(p)=i})$. Further note that as the sampling of test points is nearly uniform then $(1-\delta) s_{i,j} \leq E[t_{i,j}] \leq (1+\delta) s_{i,j}$. For notational convenience, let $m = E[t_{i,j}]$. We now break the proof into three cases, that cover the ways in which the statement of this lemma would not be true. For each case, we show with high probability the case does not occur.

Case 1: $ratio'_{i,j} \leq \frac{1}{2k^{\varphi 2} \log N}$ and $ratio_{i,j} > \frac{1-\varepsilon}{2k^{\varphi 2} \log N}$ and $ratio'_{i,j} \geq (1+\varepsilon) ratio_{i,j}$. We are going to prove the probability of this case happening is very low. If we set $t = \varepsilon m$, then using Chernoff bound, we have

$$
\begin{aligned}
\Pr[t_{i,j} \geq (1+\varepsilon)m] &\leq \exp\left(\frac{(\varepsilon m)^2}{2m + \varepsilon m}\right) && \text{[Upper Chernoff Bound]} \\
&\leq \exp\left(\frac{\varepsilon^2 (1-\delta) ratio_{i,j} s_{i,j}}{2+\varepsilon}\right) && [m \geq (1-\delta) ratio_{i,j} s_{i,j}] \\
&\leq \exp\left(\frac{\varepsilon^2 (1-\delta)(1-\varepsilon) s_{i,j}}{3(2k^{\varphi 2} \log N)}\right) && \left[ratio_{i,j} > \frac{1-\varepsilon}{2k^{\varphi 2} \log N}\right] \\
&\leq \exp\left(\frac{\varepsilon^2 (1-\delta)(1-\varepsilon) t\, k^{\varphi 2} \log^2 N}{3(2k^{\varphi 2} \log N)(3\varepsilon^2)}\right) && \left[s_{i,j} \geq \frac{t}{3\varepsilon^2} k^{\varphi 2} \log N\right] \\
&= \exp\left(\frac{(1-\delta)(1-\varepsilon) t \log N}{18}\right)
\end{aligned}
$$

Therefore, for $\delta = \varepsilon = 1/10$ and $t \geq 30$ this case cannot happen with high probability.

Case 2: $ratio'_{i,j} \leq \frac{1}{2k^{\varphi 2} \log N}$ and $ratio_{i,j} > \frac{1-\varepsilon}{2k^{\varphi 2} \log N}$ and $ratio'_{i,j} < (1-\varepsilon) ratio_{i,j}$. We can use Lower Chernoff Bound with $h = \varepsilon m$ to prove the probability of this event is very small.

$$
\begin{aligned}
\Pr[t_{i,j} \leq (1-\varepsilon)m] &\leq \exp\left(\frac{(\varepsilon m)^2}{3m}\right) \\
&\leq \exp\left(\frac{\varepsilon^2 (1-\delta) ratio_{i,j} s_{i,j}}{3}\right) && [m \geq (1-\delta) ratio_{i,j} s_{i,j}] \\
&\leq \exp\left(\frac{\varepsilon^2 (1-\delta)(1-\varepsilon) s_{i,j}}{3(2k^{\varphi 2} \log N)}\right) && \left[ratio_{i,j} > \frac{1-\varepsilon}{2k^{\varphi 2} \log N}\right] \\
&\leq \exp\left(\frac{\varepsilon^2 (1-\delta)(1-\varepsilon) t\, k^{\varphi 2} \log^2 N}{3(2k^{\varphi 2} \log N)(3\varepsilon^2)}\right) && \left[s_{i,j} \geq \frac{t}{3\varepsilon^2} k^{\varphi 2} \log N\right] \\
&= \exp\left(\frac{(1-\delta)(1-\varepsilon) t \log N}{18}\right)
\end{aligned}
$$

Therefore, for $\delta = \varepsilon = 1/10$ and $t \geq 30$ this case cannot happen with high probability.

Case 3: $ratio'_{i,j} \leq \frac{1}{2k^{02}\log N}$ and $ratio_{i,j} \geq \frac{1-e}{2k^{02}\log N}$: Since $ratio'_{i,j} = \frac{t_{i,j}}{s_{i,j}}$, in this case:

$$t_{i,j} \leq \frac{s_{i,}}{2k^{02}\log N} \tag{5.1}$$

Since $m \leq f_{i,j}(1+d)s_{i,j}$, in this case:

$$m \leq \frac{1-e}{2k^{02}\log N}(1+d)s_{i,j} \tag{5.2}$$

Thus subtracting line 5.1 from line 5.2 we conclude that:

$$t_{i,j} \geq m + \frac{(e-d+ed)s_{i,j}}{2k^{02}\log N} \tag{5.3}$$

Let $l = \frac{(e-d+ed)s_{i,j}}{2k^{02}\log N}$. We can conclude that

$$\Pr[t_{i,j} \geq m + l] \leq \exp\left(\frac{-l^2}{2m+l}\right) \quad \text{Upper Chernoff Bound}$$

$$\leq \exp\left(\frac{-l^2}{\frac{1-e}{2k^{02}\log N}(1+d)s_{i,j} + l}\right) \quad \text{Using line 5.2}$$

$$= \exp\left(\frac{-\left(\frac{(e-d+ed)s_{i,j}}{2k^{02}\log N}\right)^2}{\frac{1-e}{2k^{02}\log N}(1+d)s_{i,j} + \frac{(e-d+ed)s_{i,j}}{2k^{02}\log N}}\right)$$

$$= \exp\left(\frac{-\frac{(e-d+ed)^2 s_{i,j}}{k^{02}\log N}}{2(1-e)(1+d) + 2(e-d+ed)}\right)$$

$$\leq \exp\left(\frac{-(e-d+ed)^2 s_{i,j}}{12k^{02}\log N}\right)$$

$$= \exp\left(\frac{-(e-d+ed)^2 t\log N}{12e^2}\right) \quad \text{Substituting our lower bound on } s_{i,j}$$

Therefore, for $d = e=2 = 1=10$ and $t = 30$ this case cannot happen with high probability. □

The next case proves the how large $ratio'_{i,j}$ is when we know that $f_{i,j}$ is large.

Lemma 5.5.5. If $ratio_{i,j} > \frac{1+e}{2k^{02}\log N}$ then with high probability $ratio'_{i,j} \geq \frac{1}{2k^{02}\log N}$.

Proof. We can prove that the probability of ratio $t'_{i,j} < \frac{1}{2k'^2 \log N}$ and ratio $t_{i,j} \geq \frac{1+e}{2k'^2 \log N}$ is small. Multiplying the conditions for this case by $s_{i,j}$ we can conclude that $t_{i,j} < \frac{s_{i,j}}{2k'^2 \log N}$ and $m \leq (1-d)\frac{(1+e)s_{i,j}}{2k'^2 \log N}$. And thus $t_{i,j} \leq m - l$ where $l = \frac{(e-d-ed)s_{i,j}}{2k'^2 \log N}$. Then we can conclude that:

$$\Pr[t_{i,j} \leq m - l] \leq \exp\left(-\frac{l^2}{3m}\right) \quad \text{[Lower Chernoff Bound]}$$

$$= \exp\left(-\frac{\left(\frac{(e-d-ed)s_{i,j}}{2k'^2 \log N}\right)^2}{3m}\right)$$

$$\leq \exp\left(-\frac{\left(\frac{(e-d-ed)s_{i,j}}{2k'^2 \log N}\right)^2}{3\frac{1-e}{2k'^2 \log N}(1+d)s_{i,j}}\right)$$

$$= \exp\left(-\frac{\frac{(e-d-ed)^2 s_{i,j}}{2k'^2 \log N}}{3(1-e)(1+d)}\right)$$

$$\leq \exp\left(-\frac{(e-d-ed)^2 s_{i,j}}{12k'^2 \log N}\right) \quad [d < e - 1]$$

$$\leq \exp\left(-\frac{(e-d-ed)^2 (\frac{t}{3e^2} k'^2 \log^2 N)}{12k'^2 \log N}\right) \quad \text{[Using our lower bound on } s_{i,j}\text{]}$$

Therefore, for $d = e/2 = 1/10$ and $t \geq 30$ this case cannot happen with high probability. □

We now seek to bound the fractional weights computed by the algorithm. $D_i(p)$ denote the total weight received by a point $p \in S \cap C$ from other nodes (including the initial weight one on p). Furthermore, let $D_o(p)$ denote the total weight sent by p to all other nodes. Notice that in the final step $D_o(p) = D_i(p)$ for all p in $S \cap C$.

Lemma 5.5.6. Let $D_i(p)$ denote the total weight received by a point $p \in S \cap C$ from other nodes (including the initial weight one on p). Furthermore, let $D_o(p)$ denote the total weight sent by p to all other nodes. With high probability, for all $q \in S$, $D_i(q) \leq 1 + \frac{1+2e}{\log N} \max_{p:(p,q) \in E} D_o(p)$.

Proof. Fix the point q that redirects its weight (has outgoing arcs in G). Consider its direct predecessors $P(q) = \{p : (p,q) \in E\}$. Partition $P(q)$ as follows: $P(q) = \bigcup_{i=1,\ldots,k'} P_{c_i}(q)$,

where $P_{c_i}(q)$ is the set of points that have owed their weights into q, but c_i is actually their closest center in C. Observe the following. The point q can only belong to one donut around c_i. Due to this, $P_{c_i}(q)$ is either empty or contains a set of points in a single donut around c_i that redirect weight to q.

Fix $P_{c_i}(q)$ for some c_i. If this set is non-empty suppose this set is in the j-th donut around c_i. Conditioned on the events stated in Lemmas 5.5.3 and 5.5.5, since the points in $P_{c_i}(q)$ are undersampled, we have $|P_{c_i}(q)| \leq \frac{(1+\epsilon)2^{j-1}}{2k'^2 \log N}$. Consider any $p \in P_{c_i}(q)$. Let b_i be the number of points that p charges its weight to (this is the same for all such points). It is the case that b_i is at least $\frac{(1-\delta)2^{j-1}}{2k'}$ since p owes its weights to the points that are assigned to the center that has the most number of points assigned to it from the $(j-1)$-th donut.

Thus, q receives weight from $|P_{c_i}(q)| \leq \frac{(1+\epsilon)2^{j-1}}{2k'^2 \log N}$ points and each such point gives its weight to at least $\frac{(1-\delta)2^{j-1}}{2k'}$ points with equal split. The total weight that q receives from points in $P_{c_i}(q)$ is at most the following.

$$\frac{2k'}{(1-\delta)2^{j-1}} \sum_{p \in P_{c_i}(q)} D_0(p)$$

$$\frac{2k'}{(1-\delta)2^{j-1}} \sum_{p \in P_{c_i}(q)} \max_{p \in P_{c_i}(q)} D_0(p)$$

$$\frac{2k'}{(1-\delta)2^{j-1}} \cdot \frac{(1+\epsilon)\,2^{j-1}}{2k'^2 \log N} \max_{p \in P_{c_i}(q)} D_0(p) \qquad [|P_{c_i}(q)| \leq \tfrac{(1+2\epsilon)2^{j-1}}{2k'^2 \log N}]$$

$$\frac{1+2\epsilon}{k' \log N} \max_{p \in P_{c_i}(q)} D_0(p) \qquad [\delta \leq \tfrac{\epsilon}{2} \leq \tfrac{1}{10}]$$

Switching the max to $\max_{p:(p,q)\in E} D_0(p)$, summing over all centers $c_i \in C$ and adding the original unit weight on q gives the lemma.

□

The following crucial lemma bounds the maximum weight that a point can receive.

Lemma 5.5.7. Fix h to be a constant smaller than $\frac{\log(N)}{10}$ and $\epsilon < 1$. Say that for all $q \in S \cap C$ it is the case that $D_0(q) = h$ $D_1(q)$. Then, with high probability for any $p \in S \cap C$ it is the case that $D_1(p) \leq 1 + \frac{2h}{\log N}$.

Proof. We can easily prove this by induction on nodes. The lemma is true for all nodes that have no incoming edges in G. Now assume it is true for all nodes whose longest path that reaches them in G has length $t-1$. Now we prove it for nodes whose longest path that reaches them in G is t. Fix such a node q. For any node p such that $(p,q) \in E$, by

induction we have $D_i(p) \leq 1+\frac{2h}{\log N}$, so $D_o(p) \leq 2(1+\frac{2h}{\log N})$. By Lemma 5.5.6, $D_i(q) \leq 1+\frac{1+2e}{\log N}\max_{p:(p,q)\in E} D_o(p) \leq 1+\frac{h(1+2e)}{\log N} \leq 1+\frac{2h}{\log N} = 1+\frac{h}{\log N}+\frac{h}{\log N}\cdot\frac{2(1+2e)h+2e}{\log N} \leq 1+\frac{2h}{\log N}$.

□

Comparing Alternative Weights to Fractional Weights

It only remains to bound the cost of mapping points to the centers they contribute weight to. This can be done by iteratively charging the total cost of reassigning each node with the flow. In particular, each point will only pass its weight to nodes that are closer to their center. We can charge the flow through each node to the assignment cost of that node to its closest center, and argue that the cumulative reassignment cost bounds the real fractional assignment cost. Further, each node only has flow going through it. This will be sufficient to bound the overall cost in Lemma 5.5.9.

Lemma 5.5.8. With high probability, for every center c_i, it is the case that the estimated weight w_i^0 computed by the weighting algorithm is $(1-2e)w_i^f$ where w_i^f is the fractional weight of i.

Proof. Apply union of bounds to Lemma 5.5.3 and 5.5.5 over all i and j.

Fix a center c_i. Consider all of the points that are closest to c_i and are not undersampled. Let w_i^s denote the number of these points. All the incomming edges into c_i in G, are coming from these points; therefore based on Lemma 5.5.7, $w_i^f \leq w_i^s(1+\frac{2}{\log(N)})$. On the other hand, w_i^0 is $(1-e)$ approximation of w_i^s. Therefore, $\frac{1-e}{1+\frac{2}{\log(N)}}w_i^f \leq w_i^0 \leq (1+e)w_i^f$. Assuming that $\log N$ is sufficiently larger than e, the lemma follows.

□

Comparing Fractional Weights to Optimal

Next we bound the total cost of the fractional assignment defined by the flow. According to the graph G, any point $p \in S$ and $c_i \in C$, we let $w(p,c_i)$ be the fraction of weights that got transferred from p to c_i. Naturally we have $\sum_{c_i\in C} w(p,c_i) = 1$ for any $p \in S$ and the fractional weights $w_i^f = \sum_{p\in S} w(p,c_i)$ for any $c_i \in C$.

Lemma 5.5.9. Let f_{opt} be the optimal k-means cost on the original set S. With high probability, it is the case that:

$$\sum_{p\in S}\sum_{c_i\in C} w(p,c_i)\|p-c_i\|^2 \leq 160(1+e)f_{opt}$$

Proof. Let $f = \sum_{p \in S} \|p - c_{a(p)}\|^2$. Consider any point $p \in S$ and center c_i such that $w(p, c_i) > 0$. Let P be any path from p to c_i in G. If node p's only outgoing arc is to its closest center $c_{a(p)} = c_i$, then $P = p \to c_i$, we have $\sum_{c \in C} w(p, c)\|p - c\|^2 = \|p - c_{a(p)}\|^2$. Otherwise assume $P = p \to q_1 \to q_2 \to \cdots \to q_\ell \to c_i$. Note that the closest center to q_j is c_i. Let $D(P)$ be the fraction of the original weight of 1 on p that is given to c_i along this path according to the flow of weights. As we observed in the proof of Lemma 5.5.2, we have $\|q_1 - c_{a(p)}\| > \|q_1 - c_{a(q_1)}\| \le \|q_1 - c_{a(q_1)}\| + \|q_2 - c_{a(q_1)}\| \le \|q_2 - c_{a(q_2)}\| > \cdots > \|q_\ell - c_{a(q_\ell)}\|$. This follows because for any arc (u, v) in the graph, v is in a donut closer to $c_{a(u)}$ than the donut u is in, and v is closer to $c_{a(v)}$ than $c_{a(u)}$.

We make use of the relaxed triangle inequality for squared L_2-norms. For any three points x, y, z, we have $\|x - z\|^2 \le 2(\|x - y\|^2 + \|y - z\|^2)$. Thus, we bound $\|p - c_i\|^2$ by

$$\begin{aligned}
\|p - c_i\|^2 &= \|p - c_{a(p)} + c_{a(p)} - q_1 + q_1 - c_i\|^2 \\
&\le 2\|p - c_{a(p)} + c_{a(p)} - q_1\|^2 + 2\|q_1 - c_i\|^2 && \text{[relaxed triangle inequality]} \\
&\le 2(\|p - c_{a(p)}\| + \|c_{a(p)} - q_1\|)^2 + 2\|q_1 - c_i\|^2 && \text{[triangle inequality]} \\
&\le 8\|p - c_{a(p)}\|^2 + 2\|q_1 - c_i\|^2 && [\|p - c_{a(p)}\| \le \|c_{a(p)} - q_1\|].
\end{aligned}$$

Applying the prior steps to each q_j gives the following.

$$\|p - c_i\|^2 \le 8\left(\|p - c_{a(p)}\|^2 + \sum_{j=1}^{\ell} 2^j \|q_j - c_{a(q_j)}\|^2\right)$$

Let $P_q(j)$ be the set of all paths P that reach point q using j edges. If $j = 0$, it means P starts with point q. We seek to bound $\sum_{j=0}^{\infty} 2^j \sum_{P \in P_q(j)} D(P)\|q - c_{a(q_j)}\|^2$. This will bound the charge on point q above over all path P that contains it.

Define a weight function $D'(p)$ for each node $p \in S \cup C$. This will be a new flow of weights like D, except now the weight increases at each node. In particular, give each node initially a weight of 1. Let $D'_0(p)$ be the total weight leaving p. This will be evenly divided among the nodes that have outgoing edges from p. Define $D'_i(p)$ to be the weight incoming to p from all other nodes plus one, the initial weight of p. Set $D'_0(p)$ to be $2D'_i(p)$, twice the incoming weight.

Lemma 5.5.7 implies that the maximum weight of any point is $D'_i(p) \le 1 + \frac{4}{\log N}$. Further notice that for any q it is the case that $D'_i(q) = \sum_{j=0}^{\infty} 2^j \sum_{P \in P_q(j)} D(P)$. Letting $P(p, c_i)$ be the set of all paths that start at p to center c_i. Notice such paths correspond to how the unit weight goes to c_i. We have $w(p, c_i) = \sum_{P \in P(p, c_i)} D(P)$. Let P denote the set of all paths, $\ell(P)$ denote the length of path P (number of edges on P), and let $P(j)$ denote the jth node on path P. Thus we have the following.

$$
\begin{aligned}
& \sum_{p \in S} \sum_{c_i \in C} w(p, c_i) \|p - c_i\|^2 \\
= & \sum_{p \in S} \sum_{c_i \in C} \sum_{P \in \mathcal{P}(p, c_i)} D(P) \|p - c_i\|^2 \\
& 8 \sum_{p \in S} \sum_{c_i \in C} \sum_{P \in \mathcal{P}(p, c_i)} D(P) \Big(\sum_{j=0}^{\ell(p)-1} 2^j \|P(j) - c_{a(P(j))}\|^2 \Big) \\
= & 8 \sum_{P \in \mathcal{P}} D(P) \Big(\sum_{j=0}^{\ell(p)-1} 2^j \|P(j) - c_{a(P(j))}\|^2 \Big) \\
= & 8 \sum_{q \in S} \sum_{j=0}^{+\infty} \sum_{P \in \mathcal{P}_q(j)} 2^j D(P) \|q - c_{a(q)}\|^2 \\
= & 8 \sum_{q \in S} D_i^0(q) \|q - c_{a(q)}\|^2 \\
& \sum_{q \in S} 8\left(1 + \frac{4}{\log N}\right) \|q - c_{a(q)}\|^2 = 8\left(1 + \frac{4}{\log N}\right) f
\end{aligned}
$$

Lemma 5.5.9 follows because $k\theta \leq 1067 k \log N$, $f \leq 20 f_{opt}$ with high probability by Theorem 1 in [10].

□

Finally, we prove that finding any $O(1)$-approximation solution for optimal weighted k-means on the set (C, W^0) gives a constant approximation for optimal k-means for the original set S. Let $W^f = \{w_1^f, \ldots, w_{k'}^f\}$ be the fractional weights for centers in C. Let f_{W^f} denote the optimal weighted k-means cost on (C, W^f), and f_{W^0} denote the optimal weighted k-means cost on (C, W^0). We first prove that $f_{W^f} = O(1) f_{OPT}$, where f_{OPT} denote the optimal k-means cost on set S.

Lemma 5.5.10. Let (C, W^f) be the set of points sampled and the weights collected by fractional assignment w. With high probability, we have $f_{W^f} = O(1) f_{OPT}$.

Proof. Consider the cost of the fractional assignment we've designed. $c_i \in C$, the weight is $w_i^f = \sum_{p \in S} w(p, c_i)$. Denote the k-means cost of w by $f_w = \sum_{p \in S} \sum_{c \in C} w(p, c) \|p - c\|^2$. By Lemma 5.5.9, we have that $f_w \leq 160(1 + \varepsilon) f_{OPT}$.

Intuitively, in the following we show f_{W^f} is close to f_w. As always, we let C_{OPT} denote the optimal centers for k-means on set S. For set of points X with weights $Y : X \to R^+$ and a set of centers Z, we let $f_{(X,Y)}(Z) = \sum_{x \in X} Y(x) \min_{z \in Z} \|x - z\|^2$ denote the cost of assigning

the weighted points in X to their closest centers in C. Note that $f_{W^f} \leq f_{(C;W^f)}(C_{OPT})$ since C_{OPT} is chosen with respect to S.

$$\begin{aligned}
f_{W^f} &\leq f_{(C;W^f)}(C_{OPT}) \\
&= \sum_{c_i \in C} \Big(\sum_{p \in S} w(p;c_i) \Big) \min_{c \in C_{OPT}} \|c_i - c\|^2 && [w_i^f = \textstyle\sum_{p \in S} w(p;c_i)] \\
&= \sum_{c_i \in C} \sum_{p \in S} \min_{c \in C_{OPT}} w(p;c_i) \|c_i - c\|^2 \\
&\leq \sum_{c_i \in C} \sum_{p \in S} \min_{c \in C_{OPT}} w(p;c_i) \cdot 2(\|p - c_i\|^2 + \|p - c\|^2) && \text{[relaxed triangle inequality]} \\
&= 2f_w + 2f_{OPT} \leq 2(1+\varepsilon) f_{OPT}
\end{aligned}$$

□

Using the mentioned lemmas, we can prove the final approximation guarantee.

Proof of [Theorem 5.5.1] Using Lemma 5.5.8, we know $w_i^0 = (1 \pm 2\varepsilon) w_i^f$ for any center c_i. Let C_k^0 be k centers for $(C;W^0)$ that is g-approximate for optimal weighted k-means. Let C_{OPT}^f be the optimal k centers for $(C;W^f)$, and C_{OPT}^0 optimal for $(C;W^0)$. We have $f_{(C;W^f)}(C_k^0) \leq (1+2\varepsilon) f_{(C;W^0)}(C_k^0)$ for the reason that the contribution of each point grows by at most $(1+2\varepsilon)$ due to weight approximation. Using the same analysis, $f_{(C;W^0)}(C_{OPT}^f) \leq (1+2\varepsilon) f_{W^f}$. Combining the two inequalities, we have

$$\begin{aligned}
f_{(C;W^f)}(C_k^0) &\leq (1+2\varepsilon)^2 f_{(C;W^0)}(C_k^0) \leq (1+2\varepsilon)^2 g f_{W^0} \\
&\leq (1+2\varepsilon)^2 g f_{(C;W^0)}(C_{OPT}^f) && \text{[by optimality of } f_{W^0}\text{]} \qquad (5.4) \\
&\leq (1+2\varepsilon)^3 g f_{W^f} \leq 2g(1+2\varepsilon)^4 f_{OPT} && \text{[using Lemma 5.5.10]}
\end{aligned}$$

Let $f_S(C_k^0) = \sum_{p \in S} \min_{c \in C_k^0} \|p - c\|^2$. For every point $p \in S$, to bound its cost $\min_{c \in C_k^0} \|p - c\|^2$, we use multiple relaxed triangle inequalities for every center $c_i \in C$, and take the weighted average of them using $w(p;c_i)$.

$$
\begin{aligned}
f_S(C_k') &= \sum_{p \in S} \min_{c \in C_k'} \|p - c\|^2 \\
&= \sum_{p \in S} \sum_{c_i \in C} w(p, c_i) \min_{c \in C_k'} \|p - c\|^2 && [\textstyle\sum_{c_i \in C} w(p, c_i) = 1] \\
&\leq \sum_{p \in S} \sum_{c_i \in C} w(p, c_i) \min_{c \in C_k'} 2(\|p - c_i\|^2 + \|c_i - c\|^2) && [\text{relaxed triangle inequality}] \\
&= 2f_w + 2f_{(C, w^f)}(C_k') && [\textstyle\sum_{p \in S} w(p, c_i) = w_i^f] \\
&\leq 2f_w + 2 \cdot 32\gamma(1 + 2\varepsilon)^4 f_{OPT} && [\text{inequality (5.4)}] \\
&\leq 2 \cdot 160(1 + \varepsilon) f_{OPT} + 2 \cdot 32\gamma(1 + 2\varepsilon)^4 f_{OPT} && [\text{Lemma 5.5.9}] \\
&= O(\gamma) f_{OPT}
\end{aligned}
$$

□

5.6 Conclusion

The next natural steps in this line of research are to determine which other standard learning algorithms can be implemented relationally, and which other standard learning problems admit relational algorithms. The hope would be that such algorithms would/could eventually be implemented in software products like BigQuery ML [big].

Looking further out, one could imagine a middleware of relational algorithms incorporated into a database that application builders could use in the development of algorithms for their homegrown optimization problem (which may or may not arise from a learning application). Thus natural broader research goals are to develop generally applicable algorithmic design and analysis tools, and to determine what are the “right” relational algorithms to include in such a middleware (because the problem that they solve is a commonly useful basic building block for the development of other algorithms).

Chapter 6

Scalable Hierarchical Clustering in General Metric Spaces

6.1 Introduction

In Chapter 4 we have studied the scalable agglomerative hierarhical clustering problem. In this chapter we revisit the problem in a more general setting, where previous algorithm designs cannot be directly applied.

For ℓ_p metrics, both Abboud et al. [3] and Lavastida et al. [100] showed how to use Approximate Nearest Neighbor (ANN) data structure to improve the running time to be subquadratic for average linkage. The ANN data structure, constructed based on locality sensitive hashing (LSH), can be used to quickly find points that are close to any given point. In this work we generalize their approach and show how to implement it in any space where a hashing function like LSH exists so that an ANN data structure can be constructed. Our first key technical contribution is a new way to use ANN data structures to compute average similarity between a pair of clusters.

Many problems, however, are best captured by general metric spaces for which ANN schemes do not exist. Consider, for instance, the shortest path metric on a road graph. A general road network does not have a good ANN and therefore there is no fast average-linkage algorithm known. To overcome this we could use the latitude and longitude as the Euclidean location of points and use ℓ_2 distance to be a proxy for the road network distance. Then we have a space where LSH and an ANN data structure exists and we can leverage our algorithm. This has issues though because while the road distance and proxy are often close, natural obstacles like mountains and rivers may cause two points that are in close proximity in the Euclidean space are far apart according to the road metric.

This phenomenon is common. In several instances we have a well-behaved proxy metric that approximately preserves distances for majority of point pairs in the input. For instance, the distances between points after applying dimension reduction like Johnson–Lindenstrauss [62] or metric embedding [78]. Using these techniques, most pairwise distances are approximately preserved, but a few of the pairs have high error.

This paper investigate how one can use such proxies to improve running times of hierarchical clustering in general metric spaces. One can, of course, directly use them in place of the true metric to compute a hierarchical clustering in linear time. This approach has two drawbacks. First it leads to poor solutions as small errors can propagate and lead to a very different solution. Second, it is brittle and offers no trade-off between the quality of the solution and the running time. In contrast, we show how to use proxy metrics, ANN data structures, and the original metric in a way that allows us to keep the subquadratic running time and only pay a small cost in the accuracy of the final clustering. Our methods are applicable to any general metric for which such a proxy exists. The algorithm is evaluated with theoretical analysis and we further verify its effectiveness in several experiment settings derived from real-life scenarios. We show that despite the proxy locally distorting the original metric for some pairs, empirically our algorithm is able to make up for the distortion and identifies the approximately closest clusters to merge.

Related work. Hierarchical clustering received renewed attention in the community with a breakthrough result by [61] followed by [58] and [115]. [13] and [43] gave stronger theoretical justifications. These developments have been covered in previous chapters, mostly in Section 2.2.

The focus of our work is on improving the running time of hierarchical clustering algorithms in general metric spaces. Our main tool will be that of locality sensitive hashing (LSH), first introduced by [79], that has had many developments, see for instance [19]. However, as [54] showed formally, not all metric spaces support such Approximate Nearest Neighbor (ANN) data structures.

For hierarchical clustering specifically, [3] recently gave a way to use locality sensitive hashing to speed up average linkage algorithms. Critically, however, the methods in [3] only work for ℓ_1 spaces. We extend them to use arbitrary ANN data structures by identifying invariant properties that make speed improvements possible.

Our Contributions In this work we give new algorithms for improving the running time of hierarchical clustering algorithms.

1. We show how to use Approximate Nearest Neighbor (ANN) data structures to speed up running times of single-linkage and average-linkage algorithms.

2. In general metric spaces, when ANN data structures are not available, we show how to use proxy metrics to trade-off accuracy with running time.

3. We demonstrate such a trade-off empirically, and show how to find hierarchical clusterings within a few percent of optimal in near linear running time.

6.2 Preliminaries

Linkage-based algorithms. Both average-linkage and single linkage have been defined in Section 2.2. Here we note that the single linkage algorithm is equivalent to Kruskal algorithm for computing the Minimum Spanning Tree (MST) on the complete graph where edge weights correspond to the distance between points. We will also be interested in the approximate version of Kruskal algorithm, which selects any edge at each step whose distance is within a factor α of the smallest edge between different components. This version always returns a tree of weight at most α times optimal [37].

Approximate Nearest Neighbors. A key building block of our techniques will be the approximate nearest neighbor (ANN) data structure, as is already defined in Section 4.2.2. We recall here key facts about LSH and ANN, adopting the new notation in this chapter.

The ANN data structure relies on an existence of a family of hash functions $\mathcal{H}$ with each $h \in \mathcal{H}$ a function from S to Z, that map nearby points to the same hash value. Let $B_S(q, r)$ denote the set of points in S that has distance at most r from point $q \in S$. We have the definition of (r, gr, p_1, p_2)-sensitiveness:

Definition 6.2.1 ((r, gr, p_1, p_2)-sensitiveness). For any target distance r, point set S and any query point $q \in S$, given a constant factor g ($g > 1$), the hash function h is (r, gr, p_1, p_2)-sensitive, if for $p_1 > p_2$:

- $\forall v \in B_S(q, r)$, $\Pr[h(q) = h(v)] \geq p_1$
- $\forall v \notin B_S(q, gr)$, $\Pr[h(q) = h(v)] \leq p_2$

Families $\mathcal{H}$ satisfying these constraints are known to exist for ℓ_p norm distances in Euclidean space for $p \in (0, 2]$ [64], and are known not to exist for some metric spaces [54].

Using a family of such functions $\mathcal{H}$ we can construct a data structure for approximate nearest neighbor queries.

As we mentioned, such a family of functions $\mathcal{H}$ might not exist for a general distance function $d : S \times S \to \mathbb{R}^+$ we are considering. However, assume we have a proxy metric $d' : S \times S \to \mathbb{R}^+$ that approximates d by a factor of $b \geq 1$. Up to scaling, we assume that for any pair of points $(x, y) \in S \times S$, $\frac{1}{b} \leq \frac{d'(x,y)}{d(x,y)} \leq 1$. If there exists an ANN data structure for d', using d' as a proxy for d and applying a (r, g, p_1, p_2)-sensitive function on d' gives a (r, bg, p_1, p_2)-sensitive function for d. Then, we can combine the proxy metric d' and its LSH function to construct a data structure that works for d as well.

Definition 6.2.2. For any set of points $P \subseteq S$, $Q_P(q, r)$ is a valid (r, g)-Near-Neighbor (NN) query, if it satisfies these conditions:

- If $B_P(q, r) = \emptyset$, then $Q_P(q, r)$ returns $\emptyset$.
- If $B_P(q, r) \neq \emptyset$, then $Q_P(q, r)$ gives a point $p \neq q \in P$, such that $d(p, q) \leq g \cdot r$.

We call the point found by an approximate NN-query a *nearest neighbor*.

[79] and [64] described the way to build a data structure $D(S, r, g)$ that allows one to answer (r, g)-NN queries on any subset P using (r, gr, p_1, p_2)-sensitive hashes. Importantly, the queries run in time sublinear in $|S|$.

We let $n = |S|$ and $H(n)$ denote the time needed to construct a hash function. By the scheme described in [64], $D(S, r, g)$ is constructed by concatenating $O(\log n)$ hashes and repeating for $O(n^r \log n)$ times. Thus the construction time for $D(S, r, g)$ is bounded by $O(n^r \log^2 n H(n))$, where $r = \frac{\ln(1/p_1)}{\ln(1/p_2)} \in (0, 1)$.

Moreover, assuming it takes constant time to insert, delete or access an arbitrary point in any bucket, the query time of $Q_P(q, r)$ is $T(n, g) = O(n^r \log n)$, while the insert/delete time is $D(n, g) = O(n^r \log^2 n)$. We note that for a fixed p_1, p_2 decreases as g grows, hence making r a function of g.

6.3 Warm-up: Using ANNs to Approximate Single Linkage

In this section, we give a brief description of how an ANN data structure can be used to implement single linkage clustering.

Both [79] and [35] proposed algorithms that use (r, g)-Near-Neighbor to compute an Approximate Minimum Spanning Tree. Here we provide a modified version that is more in line with our average-linkage algorithm described in the next section.

The goal is to implement the approximate variant of Kruskal's algorithm. Initially the tree T is empty. The algorithm iteratively selects an edge e and adds to T if (1) $T \cup \{e\}$ does not form a cycle and (2) it is within a factor $(1 + \epsilon)$ of the cheapest such edge.

We now re-interpret this approach in the language of single linkage. Initially all points are clusters. In each step, clusters C_i and C_j are merged if $\min_{a \in C_i, b \in C_j} d(a,b)$ is at most $(1+\varepsilon) \min_{i' \neq j'} \min_{a \in C_{i'}, b \in C_{j'}} d(a,b)$.

Let S be the set of input points. Without loss of generality, assume the smallest interpoint distance is 1 and the aspect ratio is $D = \max_{u,v \in S} d(u,v)$. We want to find a pair of clusters to merge. We fix some $d > 0$, and proceed by merging all clusters with distance at most d between them. When no more merges are possible, we increase d by a $(1+\varepsilon)$ factor and repeat. Suppose we have a partition of S into different components (clusters) $\mathcal{C} = \{C_1, \ldots, C_m\}$. We will find edges with weights at most d between components and merge components that share such an edge. It is easy to see that this results in a $(1+\varepsilon)$-approximation of the MST.

How then to use the ANN data structure to find clusters to merge? For a threshold d, construct a data structure $D = D(S, d, g)$. Pick any component $C_i \in \mathcal{C}$, remove the points in C_i from D, and add all points in C_i to a query list. Query the first point in the list, p, using the current D. If the query returns a neighbor q, let C_j be the component containing q. Add edge (p,q) to the tree T. Then add all points in C_j to the query list and remove C_j from D. Repeat until the query returns an empty set for p, then use next point in the query list and repeat. If the query list becomes empty but there are components that haven't been queried before, pick an arbitrary component and do repeat the process again. The querying stops when D becomes empty.

The definition of the data structures immediately yields the following theorem.

Theorem 6.3.1. *With high probability, the algorithm returns a $(1+\varepsilon)$-approximate minimum spanning tree or singe-linkage tree, and has $(\frac{1}{\varepsilon} \log D \log^2 n^r H(n))$ running time.*

Proof. First we analyze the approximation ratio. Notice that after all merges for threshold d, there is no edge of weight at most d that can be added without creating a cycle. Indeed, say such an edge (p,q) exists. Assume p gets added into the query list before q. Consider the time when p is deleted from the query list, q is not in the same component with p, so since $d(p,q) \leq d$, querying p should have returned another point and p should not be deleted at this time, contradiction. Thus for threshold d, we have guarantee that the shortest edge that could be added has distance at least $\frac{d}{1+\varepsilon}$. Since all queries are (d,g)-NN queries, the edges found have weights bounded by d, giving the approximation ratio $g(1+\varepsilon)$.

Now we analyze the running time. There are $\log_{1+\varepsilon} D = Q(\frac{1}{\varepsilon} \log D)$ threshold values in total. Fix any threshold value d. The construction of data structure takes $\log^2 n \; n^r H(n)$. For every point p, every query of p either results in merging the component containing p with some other component, or deleting p from the query list and thus never querying p again for the current threshold d. So the total number of queries performed for any point in S is bounded by $O(n)$, taking $O(nQ(n,g)) = O(n^{1+r} \log^2 n)$ time in total. Any point $p \in S$

gets deleted once from the data structure, so the algorithm performs $O(n)$ deletions, taking $O(nD(n;g)) = O(n^{1+r}\log^2 n)$ time. The time spent on adding and deleting points to the query list can be in $O(n)$. Assuming $H(n) = W(n)$, the data construction time dominates the run time of algorithm, thus the run time is $O(n^r \log^2 n \log D H(n))$. □

6.4 Main Algorithm for Scalable Average Linkage

We now expand our method to show how to use ANNs to construct a provably approximate average-linkage for input using general metric distance scores. Similar to the single linkage case, the average linkage algorithm will consider a geometric series of thresholds of the form $(1+e)^k$ for $k \in [d\log De]$. For any fixed d, the algorithm merges clusters within $g(1+O(e))d$ of each other.

Like what we have seen in Chapter 4, the algorithm uses ANN to iteratively merge clusters with approximately smallest average distance for each threshold. Naively computing the distance between two clusters requires comparing the total distance of all pairs of points in the two clusters; ANN helps us avoid this quadratic dependence. However, the embedding technique described Chapter 4 will no longer apply here as the points are no longer assumed to be in a Euclidean space, while the embedding requires knowing the coordinates of the input points.

Algorithmic structure. We now detail how to use ANNs to find clusters with average linkage distance $g(1+O(e))d$ or smaller.

Let C be the current set of clusters. We represent every cluster C_i by a point in the cluster that is close to the cluster center. (If points were in Euclidean space, we would use the centroid of the cluster.) Let $f(C_i)$ be the point in C_i we use to represent this cluster, we define this formally in the next section. We will maintain a data structure of the points $f(C_i)$ for $C_i \in C$. This data structure allows for queries of the form $Q(f(C_i); r)$ for a r close to d. The query finds all pairs C_i, C_j such that $d(f(C_i), f(C_j)) \le gr$. Intuitively, the mapping f is chosen such that for any two clusters, the distance between the mapped points preserves their average distance up to some constant factor.

Unfortunately, it is hard to select f as such that $d(f(C_i), f(C_j))$ approximates $d(C_i, C_j)$ for any pair of clusters C_i, C_j. For example, consider two clusters with overlapping centers, but points are close to the center in one of them, and far away from the center in another.

To make up for the information loss, for every cluster $C \in C$, we maintain two values: one is $f(C)$ and the other is $s(C) = \frac{1}{|C|}\sum_{v \in C} d(v, f(C))$. Intuitively $s(C)$ represents the

"deviation" of points in C from the center $f(C)$. Given any two clusters $C_i, C_j \subseteq C$, we use the $d(f(C_i), f(C_j))$ and $s(C)$ to approximate the average distance $Avg(C_i, C_j)$.

At a high level, the following properties should hold for $f(\cdot)$ and $s(\cdot)$. These properties will allow us to identify clusters whose average linkage distance is small enough to merge them. (Formally, we will need slight variations on these, see the paragraph on Construction of Center and Deviation for exact details.)

1. $d(f(C_i), f(C_j)) - s(C_i) - s(C_j) \leq Avg(C_i, C_j) \leq d(f(C_i), f(C_j)) + s(C_i) + s(C_j)$.
2. $s(C_i), s(C_j) \leq c \cdot Avg(C_i, C_j)$ for a constant c.

We now describe how we find which pairs of clusters can be merged and which ones we should not consider merging. For any cluster C_i, we use the deviation term $s(C_i)$ as a filter. Observe that if $s(C_i) > cd$, then $Avg(C_i, C_j) > d$ for any C_j by the above property. This implies that no C_i with $s(C_i) > cd$ needs to be considered for merging with any other cluster.

We thus remove all such clusters from consideration. Since the deviation terms for all remaining clusters are small, property one will allow the algorithm to merge any pair of clusters where $d(f(C_i), f(C_j))$ is small compared to d. To that end, as stated before we build a nearest neighbor data structure based on the points $f(C_i)$ for all remaining clusters.

There is one problem remaining. After a merge the new cluster needs to be considered for subsequent merges. The question is how to adapt the data structure efficiently such that the query is still valid? There are two steps involved: 1) finding a representative point for the new cluster and measuring deviation, and 2) deleting the old clusters and inserting the new cluster into the data structure.

For both, we will develop a fast algorithm that constructs a randomized approximation to f and s.

Construction of center and deviation. For each cluster C in C we show how to construct $f(C)$ and $s(C)$. The purpose of this section is to establish that these quantities exist for every cluster such that they have the requisite properties. Algorithmically, this section will allow us to discover any cluster C to discard based on $s(C)$. For the kept clusters we will put their $f(C)$'s into the data structure, then use ANN queries to decide which pairs to merge.

If $|C| \leq \frac{1}{\epsilon}$, set $f(C) = \arg\min_{p \in C} Avg(C, p)$ for some constant $0 < \epsilon < 1$ to be set later. This takes $O(\frac{1}{\epsilon}|C|)$ time to determine. Now, consider C such that $|C| > \frac{1}{\epsilon}$. To begin, we calculate $(f(C), s(C))$ as follows. Sample $\frac{1}{\epsilon}$ points uniformly from C to get a sample C'. Find the point $v \in C'$ such that $\sum_{p \in C} d(v, p)$ is minimized. Set $f(C)$ equal to v. Then set $s(C) = Avg(C, f(C)) = Avg(C, v)$.

We now aim to show the properties regarding f and s.

Lemma 6.4.1. If $m = Q(\frac{1}{e}\log n)$, with high probability, $s(C) = Avg(f(C), C) \leq (1 + \frac{1}{1-e}) \min_{p \in S} Avg(p, C) \leq 2(1+e)\min_{p \in S} Avg(p, C)$ for $e \leq 0.5$.

Proof. Let c_0 be the point that minimizes $Avg(C, c_0)$ in S. Let $B_C(c_0, r)$ be a ball around c_0 of radius r, which contains the set of points in C with distance less than some r from c_0. Correspondingly let $\bar{B}_C(c_0, r)$ denote the ball with same center and radius, but including its boundaries. For any radius r_e, to simplify the notation we denote the two balls by B_e and $\bar{B}_e$ respectively. We can find an r_e such that $|B_e| < e|C|$, but $|\bar{B}_e| \geq e|C|$. We will prove any point in $\bar{B}_e$ is a good center for C. For any point $q \in \bar{B}_e$, by the triangle inequality, $\sum_{p \in C} d(p, q) \leq \sum_{p \in C} d(p, c_0) + \sum_{p \in C} d(q, c_0) \leq \sum_{p \in C} d(p, c_0) + |C| r_e$. Notice that $\sum_{p \in C} d(p, c_0) = \sum_{p \in B_e} d(p, c_0) + \sum_{p \notin B_e} d(p, c_0) \leq (1-e)|C| r_e$, since $d(p, c_0) \leq r_e$ if $p \notin B_e$. Therefore, $|C| r_e \leq \frac{1}{1-e} \sum_{p \in C} d(p, c_0)$, giving us $\sum_{p \in C} d(p, q) \leq (1 + \frac{1}{1-e}) \sum_{p \in C} d(p, c_0) \leq 2(1+e) \sum_{p \in C} d(p, c_0)$, so $Avg(C, q) \leq 2(1+e) Avg(C, c_0)$.

This implies that it is sufficient to sample a single point from $\bar{B}_e$. There are $e|C|$ points in the ball $\bar{B}_e$. Every time we sample we have e probability of getting a point p such that $Avg(C, p) \leq 2(1+e) Avg(C, c_0)$. Chernoff bounds ensure a such a point is sampled with probability at least $1 - \frac{1}{n^{W(1)}}$. □

The prior lemma immediately gives the following corollary.

Corollary 6.4.1.1. For any cluster C_i with high probability if $s(C_i) \leq 2(1+e)d$ then given any other cluster C_j, $Avg(C_i, C_j) \leq d$ for $e \leq 0.5$.

The corollary ensures that if $s(C_i) > 2(1+e)d$ then w.h.p. C_i has average linkage distance greater than d from every cluster in C. Thus, C_i need not be considered for a merge. Moreover, since average linkage distance increases as clusters get merged, we never need to consider C_i for this d.

The following lemma is a direct application of triangle inequality of average distances.

Lemma 6.4.2. For any two clusters C_i and C_j,

$$d(f(C_i), f(C_j)) - s(C_i) - s(C_j) \leq Avg(C_i, C_j) \leq d(f(C_i), f(C_j)) + s(C_i) + s(C_j)$$

The following run time lemma bounds the running time of the construction of f.

Lemma 6.4.3. For a cluster C, calculating $(f(C), s(C))$ takes $O(\frac{1}{e}|C| \log n)$ time.

Proof. It takes $O(\frac{1}{e}|C| \log n)$ time to sample $\frac{1}{e}\log n$ points, and $O(\frac{1}{e}|C| \log n)$ time to measure the average distance from C to every one of them, picking the center and calculating deviation. □

Formal algorithm. We present the formal algorithm which follows the intuition given in the previous section and the developed de nitions and lemmas.

See Algorithm 4 for the pseudocode. As stated, the algorithm considers geometrically increasing values. Fix $d_k = (1+\epsilon)^{k-1}$ and $\mathcal{C}$ to be the current set of clusters. Initially $k = 1$ and $\mathcal{C}$ has a cluster for each individual point. Let P_k be the set of centers of clusters with small deviation: $P_k = \{f(C) \mid C \in \mathcal{C}, s(C) \leq 2(1+\epsilon)d_k\}$.

The set P_k contains centers of clusters in $\mathcal{C}$ for which there may exist another cluster within average distance d_k. We maintain a data structure that supports (r, ϵ)-NN queries on P_k for a chosen $r = O(d_k)$, denoted by D_k. While P_k is not empty, we pick a point $f(C_i)$ from it and use it as a query in D_k to find a nearest neighbor. If there is no such point, $f(C_i)$ is discarded from P_k and D_k. Otherwise, if the data structures returns another point $f(C_j)$, we merge C_i and C_j, adopting the merging procedure proposed by [3]. This new point is added to P_k and D_k.

For efficiency, the pair $(f(\cdot), s(\cdot))$ is not recalculated at every merge. Rather, an update happens after a merge only if the cluster size grows significantly. We maintain a quantity $s(C)$ which denotes the size of cluster C the last time $(f(C), s(C))$ was recalculated. When we merge C_i and C_j into a new cluster C, if $|C| \geq (1+\eta)\max\{s(C_i), s(C_j)\}$ for some fixed η, the center and deviation $(f(C), s(C))$ are recalculated, and $s(C)$ is updated. Otherwise we use the center and deviation of the cluster with bigger $s(\cdot)$ value. See Algorithm 5 for details about merging.

In the next section we provide performance and running time guarantees for Algorithm 4.

6.5 Correctness and Run Time of Average Linkage Algorithm

The following theorems show that the algorithm approximates the decision of average-linkage up to a constant factor, and the algorithm runs in near-linear time. The value α is as described in the data strcutre in the preliminaries.

Theorem 6.5.1 (Running Time) Let $n = |S|$ and Δ be the aspect ratio in S. The run time of the algorithm is $O(\frac{1}{\epsilon} n^r \log^2 n \log \Delta \cdot H(n))$.

Proof. We first analyze the time needed to construct the data structures. There are $\frac{1}{\epsilon}\log\Delta$ number of threshold d's. For every d, it takes $O(n^r \log^2 n H(n))$ time to build the data structure. The total time to construct each of these data structures is bounded by $O(\frac{1}{\epsilon} n^r \log^2 n \log \Delta \cdot H(n))$.

Now we analyze the total time the algorithm spends querying the data structure or deleting a point. These queries occur to discover whether a cluster can be merged. If a cluster cannot

Algorithm 4 Main Algorithm

```
1: FastAverageLinkage(S, g, ε):
2: C ← {{p} | p ∈ S} {Make leaf clusters.}
3: f({p}) ← p, s({p}) ← 0, s({p}) ← 1 for p ∈ S {Initialize center and deviation.}
4: for k = 1, 2, ..., log_{1+ε} D do
5:    δ ← (1+ε)^{k-1}
6:    P_k ← {f(C) for C ∈ C : s(C) ≤ 2(1+ε)δ} {Filter the clusters.}
7:    r ← 5(1+ε)δ
8:    D_k ← D(P_k, r, g)
9:    while D_k is not empty do
10:      Get an arbitrary f(C_i) from P_k.
11:      Delete f(C_i) from D_k and P_k. {Maintain the data structure.}
12:      while Q_{P_k}(f(C_i), r) ≠ ∅ do
13:         f(C_j) ← Q_{P_k}(f(C_i), r)
14:         x ← (1+ε)^6(5g+4)
15:         h ← e^2/(1+x)
16:         Merge(C_i, C_j, D_k, P_k, h)
17:      end while
18:   end while
19: end for
20: Return the resulting tree.
```

Algorithm 5 Merge Algorithm

```
1: Merge(C_i, C_j, D_k, P_k, h):
2: if |C_i| + |C_j| ≤ (1+h) max{s(C_i), s(C_j)} then
3:    Update(f(C_i ∪ C_j), s(C_i ∪ C_j)) {Update only when cluster size increases by a factor.}
4:    s(C_i ∪ C_j) ← |C_i| + |C_j|
5: else
6:    if s(C_i) ≥ s(C_j) then
7:       (f(C_i ∪ C_j), s(C_i ∪ C_j)) ← (f(C_i), s(C_i))
8:       s(C_i ∪ C_j) ← s(C_i)
9:    else
10:      (f(C_i ∪ C_j), s(C_i ∪ C_j)) ← (f(C_j), s(C_j))
11:      s(C_i ∪ C_j) ← s(C_j)
12:   end if
13: end if
14: Delete f(C_j) from D_k and P_k, insert f(C_i ∪ C_j) into D_k and P_k only if s(C_i ∪ C_j) ≤ 2(1+ε)δ.
```

be merged, its corresponding point is removed from the data structure for the given threshold

d. Thus the total number of queries is $O(\frac{1}{\epsilon}(\log D)n)$. Every query takes $O(n^r \log n)$ time. In total more time is spent constructing the data structures.

Next, we analyze the total time it takes to update the data structures. By Lemma 6.4.3, every time we update the center and deviation for cluster C it takes $O(\frac{1}{\epsilon}|C|\log n)$ time. The amortized cost on every point in C is $O(\frac{1}{\epsilon}\log n)$. Pick any point $p \in S$, every time p receives an amortized cost of $O(\frac{1}{\epsilon}\log n)$, the size of the cluster p belongs to grows by a factor of $1+h$ where $h = \frac{\epsilon^2}{1+\epsilon}$. Thus the total merging and updating cost on point p is bounded by $O(\log_{1+h}(n) \cdot \frac{1}{\epsilon}\log n) = O(\frac{1}{h}\log n \cdot \frac{1}{\epsilon}\log n)$. This is bounded by $O(\frac{1}{\epsilon^3}\log^2 n)$. Taking the sum over all points in S gives a total update time cost of $O(\frac{1}{\epsilon^3}n\log^2 n)$. This is a lower order term because it is bounded by the time to construct the data structures, which is $\Omega(n)$.

□

The next theorem bounds the approximate ratio of the algorithm.

Theorem 6.5.2 (Approximation Guarantees) With high probability, Algorithm 4 gives an approximation ratio of $(5\gamma+4)(1+O(\epsilon))$ for average-linkage and completes the hierarchical clustering tree.

Our remaining goal is to prove Theorem 6.5.2. The theorem would be immediate from Lemma 6.4.2 if s and f were updated every time two clusters were merged. However, these are not always updated to ensure an efficient running time. Instead, we show the following key properties of the algorithm. Intuitively, they guarantee that if the cluster size does not grow significantly, "borrowing" the center and deviation from a cluster it is merged from will only cause the distance to be slightly distorted. The first property is critical and what allows us to derive the second and third.

Lemma 6.5.3 (Properties of Merging) During any iteration of Algorithm 5 for some fixed x, these properties always hold.

1. For any cluster $C \in \mathcal{C}$, use $\hat{C}$ to denote the subcluster of C whose center and deviation are used by C. For any cluster $C_i \in \mathcal{C}$, and another cluster $C_j \neq C_i \in \mathcal{C}$. Then we have the following two equations,

$$(1 - \tfrac{|C_i| - |\hat{C}_i|}{|C_i|})\,\mathrm{Avg}(\hat{C}_i, C_j) \quad \mathrm{Avg}(C_i, C_j) \quad (1 + \tfrac{1+\epsilon}{\epsilon} \cdot \tfrac{|C_i| - |\hat{C}_i|}{|C_i|})\,\mathrm{Avg}(\hat{C}_i, C_j) \tag{6.1}$$

and

$$(1 - \frac{|C_i| - |\hat{C}_i|}{|C_i|})(1 - \frac{|C_j| - |\hat{C}_j|}{|\hat{C}_j|}) Avg(\hat{C}_i, \hat{C}_j) \quad Avg(C_i, C_j)$$
$$(1 + \frac{1+x}{e} \frac{|C_i| - |\hat{C}_i|}{|C_i|})(1 + \frac{1+x}{e} \frac{|C_j| - |\hat{C}_j|}{|C_j|}) Avg(\hat{C}_i, \hat{C}_j) \qquad (6.2)$$

2. At the end of the iteration, no pairs of unmerged clusters have average distance less than $\frac{d}{(1+e)^2}$.
3. During this iteration, every pair of clusters we merge has distance at most $(5g+4)(1+e)^3 d$.

Notice that Theorem 6.5.2 is a corollary of the lemma. The following proof of this lemma follows logic similar to that of Lemma A.6 in [3].

Proof of [Lemma 6.5.3]

We prove this by induction. Assume we choose to merge C_i with C_j in the k-th iteration, for threshold $d = (1+e)^{k-1}$. Assume all properties hold in all iterations for $= (1+e)^t$ for $t = 0, 1, \ldots, k-2$, and all the time in the current iteration, until the merge. Specifically, we assume the algorithm finds the pair C_i and C_j correctly before merging them. Let C be the current partition of S. We will prove that after the merge of C_i and C_j, 1) the approximation ratio in property one still holds, 2) the next pair of clusters we found are close to each other, and eventually 3) there will be no clusters with average-linkage less than $\frac{d}{(1+e)^2}$ by the end of iteration for current threshold d.

Notice that the following always holds. Let $x = (1+e)^6(5g+4)$. For any other cluster $C \in C$, $Avg(C_i, C_j) \leq x \min\{Avg(C_i, C), Avg(C_j, C)\}$, and $Avg(\hat{C}_i, \hat{C}_j) \leq x \min\{Avg(\hat{C}_i, C), Avg(\hat{C}_j, C)\}$. This will be used in the proof.

Since we choose to merge C_i and C_j, by third property in Lemma 6.5.3 we must have the following by induction $Avg(C_i, C_j) \leq (1+e)^3(5g+4)d$. We also have $Avg(\hat{C}_i, \hat{C}_j) \leq (1+e)(5g+4)d$. This is because the (e, g)-NN query picked the pair $(\hat{C}_i, \hat{C}_j)$. Assume the cluster C comes from merging $C'_1, \ldots, C'_t$, where $\{C'_1, \ldots, C'_t\}$ are clusters in the partition at the end of the iteration for the previous threshold $\frac{d}{1+e}$.

We first prove $Avg(C_i, C_j) \leq x \, Avg(C_i, C)$. By property one, the average distance between C_i and any of $C'_1, \ldots, C'_t$ is at least $\frac{d}{(1+e)^3}$. By an averaging argument we have $Avg(C_i, C) = \frac{\sum_{t=1} |C'_t| Avg(C_i, C'_t)}{|C|} \geq \frac{d}{(1+e)^3}$. Letting $x = (1+e)^6(5g+4)$ gives $Avg(C_i, C_j) \leq x \, Avg(C_i, C)$. Likewise we have $Avg(C_i, C_j) \leq x \, Avg(C_j, C)$.

Next we prove $Avg(\hat{C}_i, \hat{C}_j) \leq x \, Avg(\hat{C}_i, C)$. By property one $Avg(\hat{C}_i, C) \geq (1-e)^2 Avg(C_i, C) \geq (1-e)^2 \frac{d}{(1+e)^3} > \frac{d}{(1+e)^5}$. Since $Avg(\hat{C}_i, \hat{C}_j) \leq (5g+4)(1+e)d$, the inequality holds.

Now we prove the property one still holds after the merge. Let $C'' = C_i \cup C_j$ be the new cluster. Assume first that $(C', s(C'))$ got recalculated. Then $\hat{c}'' = c'', s(C'') = |C''|$ and for any cluster C, $\mathrm{Avg}(\hat{c}'', C) = \mathrm{Avg}(C'', C)$. Both Inequality (6.1) and (6.2) are reduced to:

$$\left(1 - \frac{|C_j| - \hat{c}_j}{|C_j|}\right)\mathrm{Avg}(C'', \hat{c}'') \le \mathrm{Avg}(C'', C) \le \left(1 + \frac{1+x}{e}\cdot\frac{|C_j| - \hat{c}_j}{|C|}\right)\mathrm{Avg}(C'', \hat{c}'')$$

By induction we have

$$\left(1 - \frac{|C_j| - \hat{c}_j}{|C_j|}\right)\mathrm{Avg}(C_i, \hat{c}'') \le \mathrm{Avg}(C_i, C) \le \left(1 + \frac{1+x}{e}\cdot\frac{|C_j| - \hat{c}_j}{|C_j|}\right)\mathrm{Avg}(C_i, \hat{c}'')$$

and

$$\left(1 - \frac{|C_j| - \hat{c}_j}{|C_j|}\right)\mathrm{Avg}(C_j, \hat{c}'') \le \mathrm{Avg}(C_j, C) \le \left(1 + \frac{1+x}{e}\cdot\frac{|C_j| - \hat{c}_j}{|C_j|}\right)\mathrm{Avg}(C_j, \hat{c}'')$$

Taking the weighted average gives the inequality we want.

For the following argument, assume that $(C', s(C'))$ is not recalculated. WLOG, assumes $s(C_i) \ge s(C_j)$ and we used the center of deviation C_i, so $\hat{c}'' = \hat{c}_i$. Now, given another cluster $C \in \mathcal{C}$, we have that $\mathrm{Avg}(C, C_j) \le \mathrm{Avg}(C_i, C) + \mathrm{Avg}(C_i, C_j) \le (1+x)\mathrm{Avg}(C_i, C)$ by triangle inequality of average distances. We first prove Inequality (6.1). That is,

$$\left(1 - \frac{|C_j| - \hat{c}_j}{|C_j|}\right)\mathrm{Avg}(C'', \hat{c}'') \le \mathrm{Avg}(C'', C) \le \left(1 + \frac{1+x}{e}\cdot\frac{|C_j| - \hat{c}_j}{|C_j|}\right)\mathrm{Avg}(C'', \hat{c}'')$$

And

$$\left(1 - \frac{|C'| - \hat{c}'}{|C'|}\right)\mathrm{Avg}(\hat{c}', C) \le \mathrm{Avg}(C', C) \le \left(1 + \frac{1+x}{e}\cdot\frac{|C'| - \hat{c}'}{|C'|}\right)\mathrm{Avg}(\hat{c}', C)$$

The former inequality can be proved in the same way as the case where C'' is recalculated by taking weighted average over two inequalities on C_i and C_j. Thus we only need to prove the second one. This is done by expanding the expression of $\mathrm{Avg}(C'', \hat{c}'')$ as weighted average of $\mathrm{Avg}(C_i, \hat{c}'')$ and $\mathrm{Avg}(C_j, \hat{c}'')$ and then leverage the fact that $\mathrm{Avg}(C_j, C) \le (1+x)\mathrm{Avg}(C_i, C)$.

For upper bound,

$$\begin{aligned}
Avg(C'',C) &= \frac{|C_i|\,Avg(C_i;C) + |C_j|\,Avg(C_j;C)}{|C_i| + |C_j|} \\
&\frac{|C_i|\,Avg(C_i;C) + |C_j|(1+x)\,Avg(C_i;C)}{|C_i| + |C_j|} \\
&\frac{|C_i| + |C_j|(1+x)}{|C_i| + |C_j|}\left(1 + \frac{1+x}{e}\frac{|C_i| \quad s(C_i)}{|C_i|}\right) Avg(C_i';C) \\
&= \left(1 + \frac{x|C_j|}{|C_i| + |C_j|}\right)\left(1 + \frac{1+x}{e}\frac{|C_i| \quad s(C_i)}{|C_i|}\right) Avg(C_i';C)
\end{aligned}$$

We prove that

$$\left(1 + \frac{x|C_j|}{|C_i| + |C_j|}\right)\ \left(1 + \frac{1+x}{e}\ \frac{|C_i| \quad s(C_i)}{|C_i|}\right) \quad 1 + \frac{1+x}{e}\ \frac{|C''| \quad s(C'')}{|C''|} \qquad (6.3)$$

We have:

$$\begin{aligned}
LHS \quad & \left(1 + \frac{1+x}{e}\ \frac{|C_i| \quad s(C_i)}{|C_i|} + \frac{x|C_j|}{|C_i| + |C_j|}\ \left(1 + \frac{1+x}{e} \quad \frac{e^2}{1+x}\right)\right) \\
&= \left(1 + \frac{1+x}{e}\ \frac{|C_i| \quad s(C_i)}{|C_i|} + (1+e)\frac{x|C_j|}{|C_i| + |C_j|}\right)
\end{aligned}$$

We prove that $(1+e)\frac{x|C_j|}{|C_i|+|C_j|} \quad \frac{1+x}{e}\ \frac{|C_j|s(C_i)}{|C_i|(|C_i|+|C_j|)}$. Since $\frac{s(C_i)}{|C_i|} \quad \frac{1+x \quad e^2}{1+x}$, $\frac{1+x}{e}\ \frac{|C_j|s(C_i)}{|C_i|(|C_i|+|C_j|)}$ $\frac{1+x \quad e^2}{e}\ \frac{|C_j|}{|C_i|+|C_j|} \quad (1+e)\frac{x|C_j|}{|C_i|+|C_j|}$. Therefore,

$$\begin{aligned}
& Avg(C'',C) \quad Avg(C_i';C)\ \left(1 + \frac{1+x}{e}\ \frac{|C_i| \quad s(C_i)}{|C_i|} + (1+e)\frac{x|C_j|}{|C_i| + |C_j|}\right) \\
& Avg(C_i';C)\ \left(1 + \frac{1+x}{e}\ \frac{|C_i| \quad s(C_i)}{|C_i|} + \frac{1+x}{e}\frac{|C_j|s(C_i)}{|C_i|(|C_i| + |C_j|)}\right) \\
&= Avg(C_i';C)\left(1 + \frac{1+x}{e}\ \frac{|C_i| + |C_j| \quad s(C_i)}{|C_i| + |C_j|}\right) \\
&= Avg(C_i'';C)\left(1 + \frac{1+x}{e}\ \frac{|C''| \quad s(C_i)}{|C''|}\right)
\end{aligned}$$

For lower bound,

$$\begin{aligned}
Avg(C'', C) &= \frac{|C_i| Avg(C_i, C) + |C_j| Avg(C_j, C)}{|C_i| + |C_j|} \\
&\quad \frac{|C_i| Avg(C_i, C)}{|C_i| + |C_j|} \\
&\quad Avg(C_i, C) \quad \frac{s(C_i)}{|C_i|} \quad \frac{|C_i|}{|C_i| + |C_j|} \\
&= Avg(C_i, C) \quad \frac{s(C_i)}{|C_i| + |C_j|} \\
&= Avg(C'', C) \quad (1 \quad \frac{|C''| \quad s(C'')}{|C''|})
\end{aligned}$$

Thus we've proved that the Inequality (6.1) holds after the new merge. Using the same logic, we can prove Inequality (6.2).

For upper bound,

$$\begin{aligned}
Avg(C'', C) &= \frac{|C_i| Avg(C_i, C) + |C_j| Avg(C_j, C)}{|C_i| + |C_j|} \\
&\frac{|C_i| Avg(C_i, C) + |C_j|(1 + x) Avg(C_i, C)}{|C_i| + |C_j|} \\
&\frac{|C_i| + |C_j|(1 + x)}{|C_i| + |C_j|} \quad (1 + \frac{1 + x}{e} \quad \frac{|C_i| \quad s(C_i)}{|C_i|}) \\
&(1 + \frac{1 + x}{e} \quad \frac{|C| \quad s(C)}{|C|}) Avg(C_i, C)
\end{aligned}$$

Plugging Inequality (6.3) into the RHS gives us:

$$\begin{aligned}
Avg(C'', C) \quad &(1 + \frac{1 + x}{e} \quad \frac{|C''| \quad s(C'')}{|C''|}) \\
&(1 + \frac{1 + x}{e} \quad \frac{|C| \quad s(C)}{|C|}) Avg(C_i, C)
\end{aligned}$$

Table 6.1 Comparing the performance of different average linkage methods, New York

Sample Size	169	330	414	493	615	888	1141	1855
Aprx Ratio, mean, Proxy-AL	1.923	1.513	1.939	1.545	1.854	2.083	2.450	1.746
Aprx Ratio, mean, Proxy-Hash-AL	1.402	1.503	1.479	1.640	1.656	1.636	1.784	1.780
Aprx Ratio, 90%, Proxy-AL	3.705	2.096	3.678	1.813	2.865	3.913	3.240	2.372
Aprx Ratio, 90%, Proxy-Hash-AL	1.925	1.975	1.973	2.268	2.234	2.188	2.452	2.407
Aprx Ratio, max, Proxy-AL	21.391	13.930	28.233	20.042	54.844	44.783	228.615	42.991
Aprx Ratio, max, Proxy-Hash-AL	2.871	2.677	3.046	3.332	2.932	3.389	3.762	3.902
Global Obj, Proxy-AL	0.999	0.989	0.999	0.998	0.999	0.994	0.963	1.001
Global Obj, Proxy-Hash-AL	1.004	0.970	0.985	0.975	0.999	0.987	0.966	0.971

For lower bound,

$$\begin{aligned}
Avg(C'',C) &= \frac{|C_i| Avg(C_i,C) + |C_j| Avg(C_j,C)}{|C_i| + |C_j|} \\
&\quad \frac{|C_i| Avg(C_i,C)}{|C_i| + |C_j|} \\
&\quad Avg(c_i', c') \left(1 - \frac{|C_i| - s(C_i)}{|C_i|}\right) \left(1 - \frac{|C| - s(C)}{|C|}\right) \frac{|C_i|}{|C_i| + |C_j|} \\
&= Avg(c_i', c') \frac{s(C_i)}{|C_i|} \frac{s(C)}{|C|} \frac{|C_i|}{|C_i| + |C_j|} \\
&= Avg(c_i', c') \frac{s(C_i)}{|C_i| + |C_j|} \frac{s(C)}{|C|} \\
&= Avg(c'', c') \left(1 - \frac{|C''| - s(C'')}{|C''|}\right)\left(1 - \frac{|C| - s(C)}{|C|}\right)
\end{aligned}$$

The first property directly implies $Avg(C_i, C_j) = (1-\epsilon)^2 Avg(c_i', c_j')$ for all pairs $C_i, C_j \in \mathcal{C}$.

For the second property, assume there is (C_i^0, C_j^0) where $Avg(C_i^0, C_j^0) \le \frac{d}{(1+\epsilon)^2}$. So $Avg(c_i^0, c_j^0) \le d$. This implies that the deviations $s(c_i^0), s(c_j^0) \le 2(1+\epsilon)d$, so the two centers $f(c_i^0), f(c_j^0)$ cannot be deleted until the query returns empty set. Since $Avg(c_i^0, c_j^0) \le d, d(f(c_i^0), f(c_j^0)) \le Avg(c_i^0, c_j^0) + s(c_i^0) + s(c_j^0) \le 5(1+\epsilon)d$. Since $r = 5(1+\epsilon)d$, a valid (r,γ)-NN query can't return empty set for $f(C_i^0)$ and $f(C_j^0)$, contradiction.

For the third property, since after the merge (c'', c_j^0) is returned by a valid (r,γ)-NN query where $r = 5(1+\epsilon)d$, $Avg(c'', c_j^0) \le (1+\epsilon)(5\gamma+4)d$. Using the first property gives us $Avg(C'', C_j^0) \le (1+\epsilon)^3(5\gamma+4)d$.

□

6.6 Experiments

This section demonstrates the empirical effectiveness of our algorithms for both single linkage and average linkage clustering.

Fig. 6.1 Subsamples for New York, Bay Area and Great Lakes

Recall that one of our motivations was that ANNs may not exist for general metrics, but often times there are well-behaved proxy metrics available; this is the setting we explore in this Section.

Our single linkage and average linkage implementations are named Proxy-Hash-SL and Proxy-Hash-AL, respectively. The goals of this section are to establish the following:

- Show that both Proxy-Hash-SL and Proxy-Hash-AL have strictly sub-quadratic running times.
- Show that using single/average linkage directly on the proxy metrics results in poor quality, yet Proxy-Hash-SL and Proxy-Hash-ALhave strong performance. This will show that they new algorithms are able leverage the proxy metrics to achieve scalability, while overcoming the shortcomings of using the proxy metrics directly.
- Demonstrate that using Proxy-Hash-SL and Proxy-Hash-AL, we can nd solutions with only a small loss in quality compared with the solution found by using single/average linkage on the real metric.

Implementation Details. We implemented the algorithms in Section 6.3 and 6.4 with slight modi cations. While building the ANN data structure for querying, we set the number of concatenations and repetitions to be constant. The values of the constants are tuned according to different data sets.

To improve show the trade-off between accuracy and running time, while querying a point using the data structure, we use LSH on the proxy distance to identify the set of candidates that are nearest neighbors to the query point, but then use the real distance to pick the closest candidate. We only take a constant number of neighbors from the point's LSH bucket and pick the one with smallest real distance from the query point. If the true distance is below the threshold d_k, we merge the clusters.

Data Set	Nodes	Edges
New York	264,346	366,923
Bay Area	321,270	400,086
Great Lakes	2,758,119	3,442,829

Table 6.2 Dataset Details

Table 6.3 Ratio between total road distance of the tree and real MST, New York

Sample Size	169	330	727	1166	1825	3765	6710	14428	28985
Proxy-SL	1.760	1.304	1.554	1.784	1.412	1.805	1.753	1.883	1.511
Proxy-Hash-SL	1.035	1.016	1.021	1.024	1.027	1.030	1.029	1.022	1.024

Experiment 1: Road Maps. We use datasets from The 9th DIMACS Implementation Challenge[1]. The data files include the road networks of different cities. The data is given in graph format where nodes represent end points of roads, while edge weights represent the road lengths. Each node's latitude and longitude are provided. We choose three areas to study: New York, Bay Area, and the Great Lakes, see Table 6.2.

Based on the road network, the distance between any two points is the length of shortest path between them, termed *road distance*. This is correlated but not equivalent to the Euclidean distance calculated using lat/long values. We remark that the true metric is a general metric, and we will use the Euclidean metric as a proxy.

Table 6.4 Ratio between total real distance of the spanning tree and real MST, Seizure

Sample Size	100	200	400	800	1600
Proxy-SL	1.284	1.331	1.416	1.462	1.521
Proxy-Hash-SL	1.141	1.158	1.176	1.184	1.209

The original datasets have millions of points, so it is impractical to find all pairwise shortest-path road distances to compute the groundtruth average linkage clustering. We perform the following subsampling method: for each city, we draw a rectangle at a random position on the map. Then we take the subgraph induced by all points in this rectangle. If the subgraph is not connected, we take the biggest connected component. Figure 6.1 contains a map of all points for every city, and the boxes represent rectangles drawn for some given lat/long lengths. This allows us to get subsamples with many different sizes and study the efficiency of our methods.

Experiment 2: Random Projections of High-Dimensional Datasets. We consider Euclidean datasets in a large number of dimensions and use a Johnson-Lindenstrauss (see, for instance, Dasgupta and Gupta [62]) dimension reduction technique to reduce the number

[1] http://users.diag.uniroma1.it/challenge9/download.shtml

Table 6.5 Comparing the performance of different average linkage methods, Seizure

Sample Size	100	200	400	800	1600
Aprx Ratio, mean, Proxy-AL	1.723	1.955	2.141	2.651	3.159
Aprx Ratio, mean, Proxy-Hash-AL	1.210	1.240	1.284	1.348	1.623
Aprx Ratio, 90%, Proxy-AL	2.549	3.053	3.386	4.272	5.315
Aprx Ratio, 90%, Proxy-Hash-AL	1.438	1.456	1.504	1.567	2.120
Aprx Ratio, max, Proxy-AL	5.836	5.958	8.851	13.501	22.921
Aprx Ratio, max, Proxy-Hash-AL	1.763	1.757	1.891	1.969	2.973
Global Obj, Proxy-AL	0.993	0.991	0.993	0.986	0.990
Global Obj, Proxy-Hash-AL	1.012	1.013	1.018	1.013	1.012

of dimensions. We consider the ℓ_2 distance between the original high-dimensional points as the real distance, and the ℓ_2 distance between the projected data points as the proxy distance.

We use the seizure data set from UCI data repository.[2] The dataset has 179 dimensions and 11500 points, with every point a recording of brain activity. We project the data to 4 dimensions and take subsamples of size $[100, 200, 400, 800, 1600]$ from the original data set and test Proxy-Hash-SL and Proxy-Hash-AL. For each of the data sizes we take 5 samples, and take the average of both performance and running time on these.

Performance Metrics. We use the following metrics to measure the performance of hierarchical clustering trees. For MST (single linkage), we use the objective in the MST problem. This is the total weight of the edges chosen in the spanning tree.

For average linkage, we use two metrics. Given a sequence of cluster merges, at every merge, using the real distance, we calculate the ratio between the average linkage of the merged clusters and the minimum average linkage. We call this metric the approximation ratio. Assuming there are n points, a hierarchical clustering tree gives $n-1$ such ratios. For both datasets we show the mean, 90%-percentile and the maximum of all approximation ratios. For vanilla average linkage on the real distance, this ratio is always 1. If all $n-1$ ratios are close to 1, the hierarchical clustering tree closely resembles the tree produced by average linkage. The other metric we use is the recently developed global objective for hierarchical clustering tree introduced in Cohen-Addad et al. [58].

Running Time. The bottleneck in all computations is the time spent on computing true distances between points. For the road map data set, every computation involves finding the shortest path between a pair of points; and for seizure, it is time consuming since the original data is high-dimensional. To give an implementation and problem independent view into the performance of our methods, we report the total number of distance computations made by Proxy-Hash-SL and Proxy-Hash-AL.

[2] https://archive.ics.uci.edu/ml/datasets/Epileptic+Seizure+Recognition

(a) New York (b) Seizure

Fig. 6.2 Growth of distance computation, Proxy-Hash-AL

Results. We first compare the performance of directly using proxy distance with using Proxy-Hash-SL and Proxy-Hash-AL. Namely, we first construct a hierarchical clustering tree by running MST/average linkage on the dataset using proxy distance as the distance metric. We use “Proxy-SL” and “Proxy-AL” to refer to the results produced in this way in all tables and figures. Then we construct another tree using our implementation of Proxy-Hash-SL/Proxy-Hash-AL. Then we compare the performance of these two hierarchical clustering trees using the proposed performance metrics.

For road map dataset, Table 6.3 compares the spanning tree found by Proxy-SLand Proxy-Hash-SL. The entries show the ratio of total weights of the spanning tree to the real MST (lower is better). We report the results for $\alpha = 0.2$ in Proxy-Hash-SL. The results are quantitatively similar on all three datasets, we only show results for New York here, and postpone the other two cities to the Supplementary Material.

Table 6.1 compares the performance of hierarchical clustering tree by running average linkage directly on the proxy metric and Proxy-Hash-AL for road map in New York. The first column shows the performance metric we are using. Here “Aprx Ratio” refers to the approximation ratio, where “mean”, “90%” and “max” refers to the mean, 90%-percentile and maximum of all approximation ratios for the tree, respectively. Following that, “Proxy-AL” or “Proxy-Hash-AL” specifies which algorithm we are using for that line in the table.

Performance analysis. The new algorithms perform significantly better than building a tree directly from the proxy metric. Naively building the MST on the proxy metric has poor performance, over 50% degradation in quality. At the same time Proxy-Hash-SL gives much better results, with only 1-3% loss at $\alpha = 0.1$ to around 7-10% at $\alpha = 0.5$. This result is independent of the graph size, but lower α values in Proxy-Hash-SL lead to higher quality results. See the Supplementary Materials for more discussion about the impact of sample

size and ε on the performance of Proxy-Hash-SL. We note that in general the performance is robust to small changes in ε.

The results extend to average linkage. In Table 6.1 and 6.5, note that both Proxy-AL and Proxy-Hash-AL perform well for the global objective in Cohen-Addad et al. [58]. The degradation is negligible (about 1%) compared to the real average linkage tree. However, the statistics of approximation ratios show that on both datasets, Proxy-Hash-AL often beats Proxy-AL on both data sets in mean and 90% percentile of all approximation ratios. On seizure, the advantage is more apparent than in road maps. Especially, Proxy-Hash-AL has a significant advantage over Proxy-AL in worst-case approximation ratios. This shows Proxy-Hash-AL makes decisions which are similar to true average linkage. The quality of its decision is stable and robust against large distortion between the proxy and real distances.

Running time analysis. Next, we look at the speed of our algorithms. The main bottleneck in all of the approaches is the number of distance computations. For the naive algorithm, we must compute distances for all n^2 pairs of nodes, resulting in a quadratic running time.

Figure 6.2a and 6.2b show the growth of number of distance computations as sample size grows. In both figures, sample sizes (the x-axis) are plotted on a log-scale for better visualization. For both data sets, we draw another "benchmark" polynomial curve $y = cx^{1+r}$ to show that the growth is strictly sub-quadratic. The plots show that for road map and seizure, the r value is bounded by 0.7 and 0.3, respectively. The gain in running time might depend on the data set. The running time plots for Proxy-Hash-SL and the road maps in Bay Area and the Great Lakes can be found in Appendix B.

Chapter 7

Robust Online Correlation Clustering

7.1 Introduction

Clustering is a central unsupervised learning problem, which comes in many flavors and variants. For instance we may consider different objectives for the clustering like k-means or k-median, spectral clustering, or correlation clustering. Or we may try to perform the clustering in different computational models—batch, distributed, parallel, etc. A clustering setting that has received a lot of interest in recent years is the online paradigm, see for instance, Zhang et al. [137], Vidal [130], Lattanzi and Vassilvitskii [99], Cohen-Addad et al. [57], Guo et al. [74, 75]. In online clustering elements arrive one by one and they are irrevocably assigned to clusters at arrival time. A key property of this setting is that the resulting clustering is stable. This is particularly important in real-world systems where clusters are served directly to users or used in downstream machine learning tasks, and so modifying the clustering has a high cost.

Clustering problems like k-median and k-means are well studied in the online setting, see for instance the groundbreaking work of Meyerson [110] and Liberty et al. [103]. This chapter considers the online correlation clustering problem, which is less understood. Correlation clustering was introduced by Bansal et al. [29]. In the problem we are given as input a weighted graph with n nodes, where positive edges represent similarities and negative edges represent dissimilarities between endpoints. A partition of the nodes into clusters should minimize the sum of the negative edges contained in any cluster and the sum of positive edges between clusters. That is, minimizing disagreements in the clustering. Correlation clustering has many applications, for instance in finding clustering ensembles [34], duplicate detection [21], community mining [49], disambiguation tasks [3], and automated labelling [11, 42].

The problem is known to be NP-hard and admits approximation algorithms. For the most studied version of the problem, where the weights are restricted to be -1 or $+1$, a 2.06 approximation algorithm was shown by Chawla et al. [47]. When, in addition, the number of clusters is upper-bounded by k, a polynomial time approximation scheme was shown to exist by Giotis and Guruswami [69]. For arbitrary weights a $O(\log n)$ approximation is the best known [65].

We consider the online correlation clustering problems when edges have weights in $\{-1, +1\}$. In the online setting, nodes arrive one by one and reveal their edges to nodes that have arrived previously. The algorithm needs to make an irrevocable cluster assignment when a node arrives. The most popular form of analysis of online algorithms is *competitive analysis* where the performance of the online algorithm is compared to the best (in hindsight) optimum solution.

It is impossible to design an online algorithm for correlation clustering that has a good competitive ratio. Consider the following example. Suppose that the first two nodes to arrive are connected by a positive edge. The algorithm has to commit to either placing them in the same cluster, or in different clusters. If it places them in the same cluster then the adversary places the remaining $n-2$ nodes as follows. The nodes are split evenly between two complete cliques with all positive edges. The two initial nodes are placed in different cliques and there are no positive edges between cliques except the single edge between the two initially arriving nodes. The optimal solution in hindsight makes k disagreement by placing the cliques into two separate clusters, while the algorithm makes at least $\Omega(n)$ disagreements.

On the other hand if the algorithm places the first two nodes into different clusters then the adversary chooses the remaining instance to be a single clique of positive edges containing the two nodes. The competitive ratio of the algorithm will be unbounded as the optimal has cost 0 and the algorithm has a positive objective value of at least one.

This lower bound example (first observed in [107]) rules out finding a good algorithm via competitive analysis, since *every* algorithm will have a worst case competitive ratio of $\Omega(n)$. However, in practice, instances are typically far from worst case. To rigorously study these situations requires going beyond worst case analysis. For a rich body of work on this topic see the recent text by Roughgarden [123]. This line of work considers broadly applicable models that avoid pathological worst-case instances that are unlikely to occur in practice.

In practice, for online clustering problems, it is common to have some information about nodes that will arrive. For example when clustering temporal data we have access to last month's data and this data gives some insights on tomorrow's data. Motivated by this, it is natural to study a model where one has access to few samples of the input data representing

the past data that has been observed. Of course, if the sample is completely adversarial the problem has not become any easier. To overcome this limitation we follow previous work [90, 105] and consider a setting where the algorithm has access to a random sample of data.

The model considered is as follows. Suppose that for any (potentially adversarially chosen) input, the algorithm is given a small number of random samples from the input before the rest is revealed online. Can we prove that it is possible to design online algorithms that are near optimal? If possible, can the model be strengthened such that some of the sample is randomized while some is adversarial and we can design robust online algorithms for this more challenging setting? We answer positively to both questions in this work.

Our Contributions. We introduce the *semi-online* model for correlation clustering. In this model elements arrive online in adversarial order but beforehand a small random fraction of nodes are revealed. This setting captures situations where some part of the online input is available beforehand, a common case when analyzing temporal data. A similar semi-online model for bipartite matching is introduced by [93], where a part of the input graph is predictable and known of ine, while the remainder is unknown and arrives online.

We prove that the well-know Pivot algorithm [18] obtains a constant fraction approximation in the online setting when a small fraction of points selected at random is revealed beforehand. The key insight behind our proof is that the Pivot algorithm can use a small random sample to "sparsify" the input instance so that it is easily clusterable.

Furthermore, we show that even if a fraction of the sample is adversarially corrupted the Pivot algorithm can still recover a constant approximation. In other words, the approach is provably robust.

Finally we analyze our nding empirically and verify that the theory is predictive of empirical results. We show that even on "adversarial" inputs the Pivot algorithm obtains good performance if it has access to a small sample of the data. Without the sample, Pivot performs poorly. We further show similar behaviour can be observed when the sample is learned from historical data. These results demonstrate the connection between the theoretical model and practice.

Additional related work. The online correlation clustering problem has been studied by [107] , where they give a $O(n)$-competitive algorithm. However, their algorithm is allowed to re-assign points by merging clusters. Ailon et al. [18] show that the Pivot algorithm is 3-competitive when vertices arrive in random order.

Our setting is also closely related to the algorithm with machine learning advice setting Lykouris and Vassilvitskii [106]. In this context, Ailon et al. [17], Saha and Subramanian [125] study the correlation clustering problem and show that it is possible to obtain better performance guarantees using same-cluster queries.

7.2 Semi-Online Model for Correlation Clustering

In this paper, we consider the min-disagreements correlation clustering problem. We first introduce the offline version of the problem.

Definition 7.2.1 (Correlation Clustering) The input is a complete signed graph $G = (V, E)$ on n vertices such that the edges are partitioned into positive and negative edges $E = E^+ \cup E^-$, respectively.

The goal is to partition vertices into clusters, i.e. $V = C_1 \cup \cdots \cup C_k$, where k is arbitrary, to minimize the number of disagreements: number of positive edges with endpoints in different clusters plus the number of negative edges within each cluster. We can write this objective as: $\sum_{i<j} |\{uv \in E^+ \mid u \in C_i, v \in C_j\}| + \sum_{i=1}^{k} |\{uv \in E^- \mid u, v \in C_i\}|$.

In the online model, the vertices of the graph arrive one-by-one online. The algorithm has to maintain a partition of all arrived vertices into clusters at all times. Upon each vertex arrival, its edges (and their signs) to all previous arrivals are revealed. Then the algorithm must make an irrevocable cluster assignment for this arrival. We recall the lower bound in this setting due to Mathieu et al. [107].

Lemma 7.2.1. [107] For any n, there exists an instance of online correlation clustering where vertices arrive in adversarial order such that every algorithm has competitive ratio $\Omega(n)$.

Semi-online model. As discussed, it is impossible to obtain any positive result in the online model. Thus, we introduce the semi-online model for correlation clustering.

Definition 7.2.2 (Semi-Online Correlation Clustering). Let $G = (V, E)$ be a complete signed graph. In semi-online correlation clustering, there are two phases: an offline and online phase.

First, in the offline phase, our algorithm is given the induced subgraph $G[S]$ for some vertex set $S \subseteq V$. Then, in the online phase, the remainder of the vertices $V \setminus S$ arrive online. Upon each online arrival, the algorithm must allocate the vertex to a cluster, either starting a new singleton cluster, or assigning it to a previously defined one.

The goal is to maintain a clustering of the vertices throughout to minimize the number of disagreements.

We will consider different models for how the offline vertices S are chosen. We note that if S is chosen adversarially, then the adversary can choose S to be a graph with only negative edges, and $V \setminus S$ to be a lower bound instance of size $\Omega(|V \setminus S|)$. This gives a lower bound of $\Omega(n - |S|)$. Thus, we focus on models where S is, at least, partially chosen randomly.

The algorithm we analyze for the semi-online correlation clustering is the natural adaptation of the Pivot algorithm developed by Ailon et al. [18]. We describe it first. The algorithm takes the following steps.

Offline Pivot Algorithm

1. Initially, all vertices are unclustered.
2. Consider vertices $v \in V$ in some order.
3. If v is currently unclustered, then mark v as a pivot, and make a new cluster consisting of v and its remaining unclustered, positive neighbors. Thus, the new cluster is $\{v\} \cup \{u \in V \mid uv \in E^+, u \text{ unclustered}\}$.
4. Repeat until all vertices are clustered, and output the final clustering.

Pivot's performance critically depends on the ordering of the vertices. If the algorithm considers vertices in random order, then Pivot is a 3-approximation in expectation [18].

Pivot can be adapted to the online setting by considering the vertices in their arrival order. As we noted, this algorithm will perform poorly on adversarially ordered sequences; moreover as we will see it has poor empirical performance as well. In the semi-online model, we can take advantage of the offline phase to find better clusters for the online arriving vertices via the pivots computed in the offline phase.

Semi-Online Pivot

1. In the offline phase, randomly order the vertices S and then using the random ordering run the Pivot algorithm on G to get $C[S]$.
2. In the online phase, we continue running the Pivot algorithm on the remainder of G given the pivots and resulting clustering of $C[S]$. Note that we consider vertices in the order they arrive online.

Note that the pivots chosen in the offline phase can potentially cluster vertices in the online phase. In particular, if vertex v arrives in the online phase, and v has a positive edge to a pivot u chosen in the offline phase, then v joins u's cluster. We will show that if S is a random-sized random sample of the input, then Pivot is $(\frac{1}{e})$-competitive. Thus a small random sample is sufficient to circumvent the strong lower bounds on any online algorithm. We also show that this style of algorithm is robust, even when S is not fully random, semi-online Pivot is guaranteed to perform well.

7.3 Warm-Up: Pivoting Using a Random Sample

In this section, we consider the case where S is a random sample from V of size εn for $\varepsilon \in (0,1)$. We call this the ε-random sample model. The analysis of this setting will illustrate the main technical ideas. Later we generalize to the case where the random sample has a fraction of adversarially chosen corruptions. In this section, we prove the following.

Theorem 7.3.1. *Fix an $\varepsilon \in (0,1)$, and let S have εn samples chosen uniformly at random. Then semi-online Pivot is $(\frac{1}{\varepsilon})$-competitive in expectation.*

We also complement our upper bound with a matching lower bound.

Theorem 7.3.2. *For any $\varepsilon \in (0,1)$, every algorithm in the ε-random sample model has competitive ratio $\Omega(\frac{1}{\varepsilon})$.*

Proof. To prove the lower bounds, it suffices to consider $\varepsilon < \frac{1}{2}$. Fix a sufficiently large n such that $\frac{1}{\varepsilon} + \varepsilon n \le n$. We define a graph on n nodes. All edges are negative except those on a set L of $\frac{1}{\varepsilon}$ vertices. The edges on L form a lower bound instance of size $\frac{1}{\varepsilon}$ guaranteed by Lemma 7.2.1.

The probability that the random sample S contains no vertex from L is: $\Pr(S \cap L = \emptyset) = (1 - \frac{1}{n}) \cdot (1 - \frac{1}{n-1}) \cdots (1 - \frac{1}{n-\varepsilon n+1}) \ge (1 - \frac{1}{n})^{\varepsilon n} = \Omega(1)$. Further, conditioned on this event, the cost of any algorithm is $\Omega(\frac{1}{\varepsilon})$-competitive, because L arrives in adversarial order. □

We now prove the upper bound. Our proof will proceed in two parts. First, we bound the cost of semi-online Pivot on the random sample S. Recall that these vertices arrive in random order, so we can leverage the analysis of the original Pivot algorithm.

Second, we bound the cost of $V \setminus S$. The main difficulty here is that $V \setminus S$ arrives in adversarial order. To overcome the $\Omega(\varepsilon n)$ lower bound, we first prove a more refined lower bound on the optimal solution in terms of the positive degrees (number of positive edges incident on a vertex) in the input graph. Then we show that running semi-random Pivot on S sparsifies the remaining graph on $V \setminus S$. That is, there are few positive edges on the nodes not adjacent to a pivot in the sample. Finally, we relate the cost of Pivot to the positive degrees to complete the proof.

7.3.1 Pivot Preliminaries

Throughout this section, let OPT denote the number of disagreements made by the optimal offline algorithm on G and ALG denote the cost of semi-online Pivot. Note that OPT is a fixed quantity, while ALG depends on the random choices of S and its order.

We begin with the concept of *bad triangles*, which we use to lower bound OPT and upper bound ALG.

Definition 7.3.1 (Bad Triangle). A bad triangle in G is a triple of vertices $t = ijk$ such that two edges among them are positive, and the remaining edge is negative. We denote the set of bad triangles in G.

By case analysis one can check that any clustering must make at least one disagreement in a bad triangle. Similarly, the Pivot algorithm makes a disagreement on ij if and only if there exists a bad triangle ijk of unclustered vertices such that k is chosen as pivot. This motivates the definition for any bad triangle t of the event:

$$A_t = \{\text{one vertex of } t \text{ is chosen as a pivot while all three are unclustered}\}$$

By linearity of expectation, we can write our expected cost as:

$$E[ALG] = \sum_{t \in T} P(A_t). \tag{7.1}$$

We emphasize that this expression holds for any distribution of arrivals. The only difference is the probabilities of the A_t's. We will re-use the same notations and expression when we consider a random sample with corruptions later. However, for the remainder of this section, all probabilities are with respect to the random sample model unless otherwise noted.

To formalize the two main steps in our proof sketch, we decompose the above quantity into two parts: one for S and one for $V \setminus S$. To this end, we define for each bad triangle t the events $A_t^S = \{\text{one vertex of } t \text{ is chosen as pivot in } S \text{ while all three are unclustered}\}$ and $A_t^{V \setminus S} = \{\text{one vertex of } t \text{ is chosen as pivot in } V \setminus S \text{ while all three are unclustered}\}$.

We can re-write: $E[ALG] = cost(S) + cost(V \setminus S)$; where $cost(S) = \sum_{t \in T} P(A_t^S)$ and $cost(V \setminus S) = \sum_{t \in T} P(A_t^{V \setminus S})$. We will analyze each of these two terms separately. In particular, we first show that $cost(S) = O(OPT)$ by relating to Pivot in random order. Then we show $cost(V \setminus S) = O(\frac{1}{\varepsilon})OPT$ by relating to the positive degrees. Combining these two bounds with the above expression for $E[ALG]$ completes the proof of Theorem 7.3.1.

7.3.2 Bounding Cost of S

We show $cost(S) = O(OPT)$. Because the vertices of S are a random subset of V and we can analyze them in random order, the first arrivals of the ε-random sample model are distributed identically to the first εn arrivals in the model where all vertices arrive in random order. We rely on the next theorem from Ailon et al. [18], which bounds the cost of Pivot when vertices arrive in random order.

Theorem 7.3.3. [18] If vertices arrive in random order, then Pivot is 3-competitive in expectation.

Now we use that Equation (7.1) holds for both the random sample model and the random order model. We denote the former distribution with a subscript S and the latter with a subscript R. Let RAND denote the cost of running Pivot in the random order model. Then applying Equation (7.1) and Theorem 7.3.3, we have $\text{cost}(S) = \sum_{t \in T} P_S(A_t^S) = \sum_{t \in T} P_R(A_t^S) \leq \sum_{t \in T} P_R(A_t) = E_R[\text{RAND}] \leq 3\text{OPT}$.

The second equality follows from the observation that the first t arrivals in the random sample model are distributed identically to the first t arrivals in the random order model, and the event A_t^S does not depend on the arrival order of $V \setminus S$. Thus for any bad triangle t, $P_S(A_t^S) = P_R(A_t^S)$. To understand the latter probability, in the random order model we use the convention that the ordered set S is the first en arrivals.

7.3.3 Bounding Cost of $V \setminus S$

We show $\text{cost}(V \setminus S) = O(\frac{1}{e})\text{OPT}$. We begin with a lower bound on OPT in terms of the positive degrees. Because every clustering must make at least one disagreement on each bad triangle, we can interpret a clustering as covering all bad triangles using edges (which are the disagreements that this clustering makes.) Further, every bad triangle has two positive edges, so we show that the number of bad triangles that any edge can cover is proportional to the positive degrees of its endpoints.

Lemma 7.3.4. Fix any clustering C and let E^* denote the set of edges that C makes a disagreement on. Then $|T| \leq \sum_{ij \in E^*} d^+(i) + d^+(j)$, where $d^+(\cdot)$ denotes the positive degree of a vertex.

Proof. C must make at least one disagreement on each bad triangle. It follows that C must cover each bad triangle with at least one edge in E^*. It suffices to show that each $ij \in E^*$ can be in at most $d^+(i) + d^+(j)$ bad triangles. To see this, we consider two cases. If $ij \in E^+$, then each bad triangle including ij must also have two positive edges ik and jk for some k. There can be at most $\min(d^+(i), d^+(j))$ such k. Otherwise, $ij \notin E^+$. Then each bad triangle including ij must have one other positive edge – either ik or jk for some k. There can be at most $d^+(i) + d^+(j)$ such k. □

To understand the utility of the above lemma, it is informative to upper bound $d^+(i)$ by the max positive degree, say d. Then Lemma 7.3.4 gives $|T| = O(d)\text{OPT}$. Further, by Equation (7.1), the expected cost of Pivot (for any arrival distribution) is $\sum_{t \in T} P(A_t) \leq |T|$. Combining these two inequalities gives that Pivot is $O(d)$-approximation even in

adversarial order. In the worst case, we have $\mathbb{E}[\text{cost}(V \setminus S)] = \Omega(n)$ (as in the standard $\Omega(n)$ lower bound instance.)

To overcome this, note that some nodes $v \in V \setminus S$ are "pre-clustered" by running Pivot on S. This occurs when a vertex $v \in V \setminus S$ has a positive edge to a pivot $s \in S$. Bad triangles containing such vertices do not contribute to $\text{cost}(V \setminus S)$, so it suffices to consider the remaining subgraph of vertices in $V \setminus S$ that are not pre-clustered. We show that this random subgraph has small positive degrees in expectation, so applying Lemma 7.3.4 to this sparse random subgraph allows us to bound $\text{cost}(V \setminus S)$.

We can now show our key structural lemma about sparsification. The main idea of the proof is that if a vertex has high positive degree, then it is likely that one of its positive neighbors becomes a pivot in S.

Lemma 7.3.5. *For any $v \in V$, define the random variable N_v to be 0 if v is clustered by running Pivot on S (i.e. $v \in S$ or v has a positive edge to a pivot in S) or $N_v = |\{\text{unclustered positive neighbors of } v \text{ in } V \setminus S\}|$ otherwise. Then $\mathbb{E}[N_v] = O(\frac{1}{\varepsilon})$.*

Proof. Fix $v \in V$ and $k \geq 0$. We first upper bound $\mathbb{P}(N_v \geq k)$. Define the event $C_i = \{v \text{ is unclustered and has at least } k \text{ positive unclustered neighbors after the } i\text{th arrival in } S\}$. Then $\mathbb{P}(N_v \geq k) = \mathbb{P}(C_{|S|} \mid C_{|S|-1}, \ldots, C_1) \cdots \mathbb{P}(C_1)$. For all i, we have:

$$\mathbb{P}(C_i \mid C_{i-1}, \ldots, C_1) \leq 1 - \mathbb{P}(v \text{ clustered by } i\text{th arrival} \mid C_{i-1}, \ldots, C_1) \leq 1 - \frac{k+1}{n-(i-1)} \leq 1 - \frac{k}{n}$$

where in the second inequality, we use the fact that conditioned on C_{i-1}, v has at least k unclustered neighbors before the ith arrival.

Recalling $|S| = \varepsilon n$, we have $\mathbb{P}(N_v \geq k) \leq (1-\frac{k}{n})^{\varepsilon n} \leq \exp(-\varepsilon k)$. Using this tail bound, we can bound the expectation of N_v: $\mathbb{E}[N_v] = \sum_{k=0}^{\infty} \mathbb{P}(N_v \geq k) \leq \sum_{k=0}^{\infty} \exp(-\varepsilon k) \leq O(\int_0^{\infty} \exp(-\varepsilon k)\,dk) = O(\frac{1}{\varepsilon})$. □

To finish our bound on $\text{cost}(V \setminus S)$, it remains to combine Lemma 7.3.4 with Lemma 7.3.5. To this end, we let G' denote the random subgraph of G induced by all vertices in $V \setminus S$ that are not clustered by the pivots chosen in S. The key properties of G' are:

- Let T' be the set of all bad triangles in G'. We have $\sum_{t \in T} \mathbb{1}_{A_t^{V \setminus S}} \leq |T'|$. To see this, note that if $A_t^{V \setminus S}$ occurs, then no vertex of t is pre-clustered by S, so $t \in T'$.
- The positive degree in G' of a vertex v is exactly N_v (as defined in Lemma 7.3.5.) Note that here we use the fact that if v is clustered by running Pivot on S, $N_v = 0$, since v is not a vertex in G'.

- Let E^0 denote the edge set of G^0 and E the disagreements made by OPT on the whole graph G. Recall that OPT is de ned on G, so E is invariant. Then OPT induces a clustering of G^0 that makes disagreements $E^0 \setminus E$.

Using the above two lemmas and three properties, we conclude $\mathrm{cost}(V \cap S) \mid T] \le E[\sum_{ij \in E^0 \setminus E} d^+_{G^0}(i) + d^+_{G^0}(j)] \le \sum_{ij \in E} EN_i + EN_j = O(\frac{1}{\epsilon})OPT$. This completes the proof of Theorem 7.3.1.

7.4 Random Sample with Adversarial Corruptions

We now consider the case where S is a random sample from V with adversarial corruptions. Our goal is to show that even with adversarial additions, S, pivot has strong approximation guarantees. We assume that S is generated by the following procedure, parameterized by $\epsilon \in (0,1)$ and $\alpha \in (0,\epsilon)$:

ϵ-Random Sample with α-Corruption

1. Given a complete signed graph $G = (V,E)$, an adversary chooses a fixed set A of αn vertices.
2. Then a uniform random subset R of size $(\epsilon - \alpha)n$ is drawn from $V \setminus A$.
3. Our algorithm is given $S = R \cup A$ in the offline phase.

We emphasize the order of events: first the adversary chooses A, and then a random sample R is drawn. In particular, the choice of A does not depend on R. Also, our algorithm is given S in the offline phase, but critically it is unaware of which vertices belong to A or R.

We briefly discuss some slight variants of our model and their tractability. Sampling $(\epsilon - \alpha)n$ from $V \setminus A$ rather than from V itself is mainly for technical convenience; the latter model admits similar theoretical guarantees, because $R \cup A$ is close to uniform on $V \setminus A$ if R is a uniform random sample of V. However, if we change the order of events, so first sample R, and then allow an adversary to remove/replace αn vertices from the sample, then the competitive ratio degrades to $\Omega(n)$. To see this, consider an input graph consisting of a lower-bound instance of size αn guaranteed by Lemma 7.2.1 and all other edges are negative. The adversary can guarantee that the lower-bound instance is always outside the sample and thus arrives in adversarial order.

The main result of this section is that the corruptions do not degrade the performance of semi-online pivot; in particular, the algorithm performs as if it had an uncorrupted sample of size $(\epsilon - \alpha)n$ (under the mild assumption that α is at most a constant fraction of ϵ).

Theorem 7.4.1. For $\varepsilon \in (0,1)$ and $\alpha \in (0,\varepsilon)$, semi-online pivot in the ε-random sample with α-corruption model is $O(\frac{\varepsilon}{(\varepsilon-\alpha)^2})$-competitive in expectation.

Similar to the random sample model, we have an almost matching lower bound for the ε-random sample with α-corruption model under the mild assumption that the corruption is at most half of the whole sample. The proof of this theorem is similar to Theorem 7.3.2.

Theorem 7.4.2. For any $\varepsilon \in (0,1)$ and $\alpha \in (0,\frac{\varepsilon}{2})$, every algorithm in the ε-random sample with α-corruption model has competitive ratio $\Omega(\frac{1}{\varepsilon-\alpha})$.

Proof. It suffices to consider $\varepsilon - \alpha < \frac{1}{2}$. Then $\varepsilon \in (0,\frac{3}{4})$. Consider any sufficiently large n such that $\frac{1}{\varepsilon-\alpha} + \varepsilon n \le n$. Then we define the graph on n vertices. All edges are negative except those on a set L of $\frac{1}{\varepsilon-\alpha}$ vertices. The edges on L are a lower bound instance of size $\frac{1}{\varepsilon-\alpha}$ guaranteed by Lemma 7.2.1.

The adversary chooses the corrupted nodes of the sample to be those in L. Let R be the corrupted random sample given to the algorithm. Then the probability that R contains no vertex from L is:

$$\Pr(R \cap L = \emptyset) = \left(1 - \frac{1/(\varepsilon-\alpha)}{(1-\alpha)n}\right) \cdots \left(1 - \frac{1/(\varepsilon-\alpha)}{(1-\alpha)n - (\varepsilon-\alpha)n + 1}\right)$$
$$\left(1 - \frac{1/(\varepsilon-\alpha)}{n - \varepsilon n}\right)^{(\varepsilon-\alpha)n} = \Omega(1)$$

Further, conditioned on this event, the cost of any algorithm is $\Omega(\frac{1}{\varepsilon-\alpha})$-competitive, because L arrives in adversarial order. □

Proof Overview of Theorem 7.4.1: The proof of Theorem 7.4.1 has the same structure as Theorem 7.3.1; in particular, we split the cost of semi-online Pivot into an offline- and online phase. The offline- and online phases roughly correspond to S and $V \setminus S$, respectively. However, the corruptions introduce new challenges in both cases.

For the offline phase, we can no longer argue that S is distributed as the first εn arrivals in the random order model due to the adversarially chosen part of S. Instead, we prove a generalization of Theorem 7.3.3 via dual fitting that allows us to handle more general arrival distributions. Roughly, to bound the cost of the offline phase, it suffices to show that if A_t occurs in the offline phase, then each vertex x of A_t has reasonable probability to be the pivot that "causes" A_t.

For the online phase, we still argue by Lemma 7.3.4 that it suffices to show that the remaining vertices in the online phase have small positive degree. Here, we prove that even with adversarial corruptions, running Pivot in the offline phase still sparsifies the remaining graph.

7.4.1 Analysis: Proof of Theorem 7.4.1

Throughout this section, let Opt denote the number of disagreements made by the optimal of ine algorithm on G and Alg denote the cost of semi-online Pivot. Note that Opt is a xed quantity, while Alg depends on the random choices of S and its order. Further, all probabilities are taken with respect to the random sample with a -corruption model.

By Equation (7.1), we can write our cost as:

$$EAlg = \sum_{t \in T} P(A_t):$$

Previously, we split this sum into the contributions by S and $V \setminus S$. Now, we consider a slightly different partition. To this end, we de ne the (random) ordered set S' of the rst $\frac{e-a}{2}n$ arrivals in S. Recall that $A_t^{S'} = f$ one vertex of t is chosen as pivot in S' while all three are unclustered g and analogously for $A_t^{V \setminus S}$. Then we can re-write:

$$EAlg = cost(S') + cost(V \setminus S');$$

where $cost(S') = \sum_{t \in T} P(A_t^{S'})$ and $cost(V \setminus S') = \sum_{t \in T} P(A_t^{V \setminus S'})$. Thus, now our of ine phase corresponds to S', and our online phase to $V \setminus S'$. We will show that $cost(S') = O(\frac{1}{e-a})Opt$ and $cost(V \setminus S') = O(\frac{e}{(e-a)^2})Opt$. Combining these two bounds with the above expression for $EAlg$ completes the proof of Theorem 7.4.1.

Bounding Bad Triangles in the Cost of S'

We show that $cost(S') = O(\frac{1}{e-a})Opt$. Recall that S' is part of the of ine phase which arrives in random order. For any vertex $i \in V$, we de ne the event $A_i^{S'}$ analogously as $A_t^{S'}$ for triangle t. That is, $A_i^{S'} = f i$ is chosen as pivot in $S' g$. We begin with a technical lemma that analogous to the dual tting analysis of Pivot in random order.

Lemma 7.4.3. Let $c > 0$. Suppose for all $t \in T$ and $i \in t$, we have $P(A_i^{S'} \mid A_t^{S'}) \leq c$. Then $\sum_{t \in T} P(A_t^{S'}) \leq \frac{1}{c}Opt$.

Proof. Consider the primal-dual pair of linear programs P and D:

$$P = \min_x f \sum_{e \in E} x_e \mid x_{ij} + x_{jk} + x_{ki} \geq 1 \; 8i\,j\,k \in T; x \geq 0g:$$

$$D = \max_y f \sum_{t \in T} y_t \mid \sum_{t \in T: e \in t} y_t \leq 1 \; 8e \in E; y \geq 0g:$$

We observe that $x_e = 1_{\text{Opt makes a disagreement on } e}$ for all e is feasible for P with objective value Opt. Thus, $Opt(P) \leq Opt$. Further, by strong duality, $Opt(D) = Opt(P) \leq Opt$.

We now exhibit a feasible solution to D. For all $t \in T$, take $y_t = cP(A_t^{S'})$. It suffices to show that this setting of y is feasible for D, because then $\sum_{t \in T} P(A_t^{S'}) \leq Opt$. It is immediate that $y \geq 0$, so we check for each $ijk \in E$:

$$\sum_{k \in V : ijk \in T} cP(A_{ijk}^{S'}) \geq \sum_{k \in V : ijk \in T} P(A_k^{S'} \mid A_{ijk}^{S'}) P(A_{ijk}^{S'}) = \sum_{k : ijk \in T} P(A_k^{S'} \cap A_{ijk}^{S'});$$

where we observe that the events $A_k^{S'} \cap A_{ijk}^{S'}$ are disjoint for all k with $ijk \in T$, and each such event implies that Pivot makes ij a disagreement. We conclude:

$$\sum_{k \in V : ijk \in T} cP(A_{ijk}^{S'}) \geq P(\text{Pivot makes } ij \text{ a disagreement}) \geq 1.$$

Thus, y is feasible for D. □

Note that we can recover the proof of Theorem 7.3.3 using the above lemma. Take all probabilities with respect to the random order model, and $S' = V$. Then $P(A_i^V \mid A_t^V) \leq \frac{1}{3}$ for all $t \in T$ and $i \in t$. This is because each vertex in t has equal probability of arriving first among the three in the random order model. This implies the expected cost of Pivot in the random order model is at most $3 \cdot Opt$.

In light of the above lemma, we study the probabilities $P(A_i^{S'} \mid A_t^{S'})$ for $t \in T; i \in t$. Roughly, we want to show that the probability of each vertex arriving first among the three is not too small. The case to keep in mind is if some vertices of a bad triangle are adversarial and others are not. The next lemma bounds shows that the probability than the next arrival in S' is a particular vertex does not vary much between adversarial and random vertices.

Lemma 7.4.4. Let $r \geq |S'|$. Fix an ordered prefix $p = (p_1, \ldots, p_{r-1})$ of the first $r-1$ arrivals in S'. Then for any $i \neq p$, we have:

$$\frac{e-a}{2} \cdot \frac{1}{en-(r-1)} \leq P(r\text{th arrival is } i \mid \text{first } r-1 \text{ arrivals are } p) \leq \frac{1}{en-(r-1)}.$$

Proof. Fix $i \neq p$ and let P be the event $P = \{\text{first } r-1 \text{ arrivals are } p\}$. Noting that $i \in S$ is necessary for i to be the rth arrival, we have:

$$P(r\text{th arrival is } i \mid P) = P(r\text{th arrival is } i \mid P, i \in S) P(i \in S \mid P).$$

Conditioned on P and $i \in S$, the rth arrival is a uniform random vertex from $S \setminus P$, so:

$$P(r\text{th arrival is } i \mid P, i \in S) = \frac{1}{|S| - |P|} = \frac{1}{\varepsilon n - (r-1)}.$$

To analyze the second term $P(i \in S \mid P)$, we consider two cases. In the first case $i \in A$, so $P(i \in S \mid P) = 1$. Otherwise, $i \in V \setminus A$, so conditioned on P, $i \in S$ if and only if $i \in R \setminus \{p_1, \ldots, p_{r-1}\}$. Thus we can lower-bound $P(i \in S \mid P) \geq \frac{(\varepsilon - \alpha)n - r}{n} \geq \frac{\varepsilon - \alpha}{2}$, where we recall $r \leq |S^0| = \frac{(\varepsilon - \alpha)n}{2}$. We conclude $\frac{\varepsilon - \alpha}{2} \leq P(i \in S \mid P) \leq 1$. Combining our expressions for both terms gives the desired result. □

Now we are ready to lower bound $P(A_i^{S^0} \mid A_t^{S^0})$ by considering the prefixes of arrivals where $A_t^{S^0}$ occurs.

Lemma 7.4.5. *For any $t \in T$ and $i \in t$, $P(A_i^{S^0} \mid A_t^{S^0}) \geq \frac{\varepsilon - \alpha}{5}$.*

Proof. Fix $t \in T$ and $i \in t$. For $1 \leq r \leq |S^0|$, we define the event

$$A_t^r = \{\text{one vertex of } t \text{ is chosen as pivot in the } r\text{th arrival while all three are unclustered}\}$$

Note that the events A_t^r for all $1 \leq r \leq |S^0|$ partition $A_t^{S^0}$. Then by the law of total probability:

$$P(A_i^{S^0} \mid A_t^{S^0}) = \sum_{r=1}^{|S^0|} P(A_i^{S^0} \mid A_t^{S^0}, A_t^r) P(A_t^r \mid A_t^{S^0}) = \sum_{r=1}^{|S^0|} P(r\text{th arrival is } i \mid A_t^r) P(A_t^r \mid A_t^{S^0}).$$

Because $\sum_{r=1}^{|S^0|} P(A_t^r \mid A_t^{S^0}) = 1$, it suffices to lower bound $P(r\text{th arrival is } i \mid A_t^r) \geq \frac{\varepsilon - \alpha}{5}$ for all $1 \leq r \leq |S^0|$.

To this end, we fix any $1 \leq r \leq |S^0|$, and let $t = ijk$. Noting that the events that the rth arrival is i, j, or k partition A_t^r, we have:

$$P(r\text{th arrival is } i \mid A_t^r) + P(r\text{th arrival is } j \mid A_t^r) + P(r\text{th arrival is } k \mid A_t^r) = 1$$

$$P(r\text{th arrival is } i \mid A_t^r) = \left(1 + \frac{P(r\text{th arrival is } j \mid A_t^r)}{P(r\text{th arrival is } i \mid A_t^r)} + \frac{P(r\text{th arrival is } k \mid A_t^r)}{P(r\text{th arrival is } i \mid A_t^r)}\right)^{-1}.$$

It remains to upper bound the ratio $\frac{P(r\text{th arrival is } u \mid A_t^r)}{P(r\text{th arrival is } v \mid A_t^r)}$ for $u, v \in t$. To this end, we define the set $\mathcal{P}$ of all ordered prefixes p of $r-1$ arrivals such that after running Pivot on p, all vertices in t are uncovered. In particular, A_t^r occurs if and only if the first $r-1$ arrivals is some $p \in \mathcal{P}$

and the rth arrival is in t. Then for any $u \in t$:

$$\begin{aligned}
P(r\text{th arrival is } u \mid A_t^r) &= \frac{P(r\text{th arrival is } u; A_t^r)}{P(A_t^r)} \\
&= \frac{\sum_{p \in P} P(r\text{th arrival is } u;\ r\text{st } r-1 \text{ arrivals are } p)}{P(A_t^r)} \\
&= \frac{\sum_{p \in P} P(r\text{th arrival is } u \mid r\text{st } r-1 \text{ arrivals are } p) P(r\text{st } r-1 \text{ arrivals are } p)}{P(A_t^r)}.
\end{aligned}$$

Now consider $u, v \in t$. Applying the upper- and lower bounds given by Lemma 7.4.4, respectively, we have:

$$P(r\text{th arrival is } u \mid A_t^r) \quad \frac{1}{en - (r-1)} \sum_{p \in P} \frac{P(r\text{st } r-1 \text{ arrivals are } p)}{P(A_t^r)};$$

$$P(r\text{th arrival is } v \mid A_t^r) \quad \frac{e-a}{2} \frac{1}{en - (r-1)} \sum_{p \in P} \frac{P(r\text{st } r-1 \text{ arrivals are } p)}{P(A_t^r)};$$

which taken together imply $\frac{P(r\text{th arrival is } u \mid A_t^r)}{P(r\text{th arrival is } v \mid A_t^r)} \quad \frac{2}{e-a}$. We conclude:

$$P(r\text{th arrival is } i \mid A_t^r) \quad (1 + \frac{2}{e-a} + \frac{2}{e-a})^{-1} = \frac{e-a}{4+e-a} \quad \frac{e-a}{5}.$$

□

Composing Lemma 7.4.3 and Lemma 7.4.5 gives $cost(S^0) = \sum_{t \in T} P(A_t^{S^0}) \quad \frac{5}{e-a} Opt = O(\frac{1}{e-a}) Opt$.

Bounding Bad Triangles in the Cost of $V \setminus S^0$

Here we show that $cost(V \setminus S^0) = O(\frac{e}{(e-a)^2}) Opt$. Our strategy again is to show that Pivot on S^0 sparsifies the remaining graph, so every arrival v in S^0 has a small number of positive unclustered neighbors. Then we apply Lemma 7.3.4. The next lemma is analogous to Lemma 7.3.5, but the estimates are slightly more involved due to the more complex arrival distribution.

Lemma 7.4.6. For any $v \in V$, define the random variable N_v to be 0 if v is clustered by running Pivot on S^0 (i.e. $v \in S^0$ or v has a positive edge to a pivot in S^0) or $N_v = |\{$unclustered positive neighbors w of v in $V \setminus S^0\}|$ otherwise. Then $E[N_v] = O(\frac{e}{(e-a)^2})$.

Proof. Fix $v \in V$ and $k \quad 0$. We first upper bound $P(N_v \quad k)$. Define the event $C_i = \{v$ is unclustered and has at least k positive unclustered neighbors after the ith arrival in $S^0\}$. Then

$P(N_v = k) = P(C_{|S'|} \mid C_{|S'|-1}, \ldots, C_1) \cdots P(C_1)$. For all i, we have:

$$P(C_i \mid C_{i-1}, \ldots, C_1) = 1 - P(v \text{ clustered by } i\text{th arrival} \mid C_{i-1}, \ldots, C_1).$$

It is convenient to describe the distribution of the ith arrival in S' as follows: Suppose the first $i-1$ arrivals consist of random arrivals $R' = V \setminus A$ and adversarial arrivals $A' = A$. First, we decide whether the ith arrival will be random (from $V \setminus A$) or adversarial (from A.) It is random with probability $\frac{(\varepsilon - \alpha)n - |R'|}{\varepsilon n - i + 1} \geq \frac{(\varepsilon - \alpha)n - |R'|}{\varepsilon n} \geq \frac{\varepsilon - \alpha}{2\varepsilon}$, using the fact that $|R'| + |S'| = (\varepsilon - \alpha)n/2$. Similarly, the ith arrival is adversarial with probability $\frac{\alpha n - |A'|}{\varepsilon n - i + 1} \geq \frac{\alpha n - |A'|}{\varepsilon n}$. After deciding whether the ith arrival will be random or adversarial, then we choose the arrival by drawing a uniform random vertex from $(V \setminus A) \setminus R'$ or $A \setminus A'$, respectively.

Now we lower bound $P(v \text{ clustered by } i\text{th arrival} \mid C_{i-1}, \ldots, C_1)$. Conditioned on C_{i-1}, v has at least k unclustered positive neighbors before the ith arrival. Let r and a denote the number of unclustered positive neighbors of v in $V \setminus A$ and A, respectively. Note that $r + a \geq k$. Then we compute:

$$\begin{aligned} P(v \text{ clustered by } i\text{th arrival} \mid C_{i-1}, \ldots, C_1) &\geq \frac{\varepsilon - \alpha}{2\varepsilon}\frac{r}{n} + \frac{\alpha n - |A'|}{\varepsilon n}\frac{a}{\alpha n - |A'|} \\ &= \frac{\varepsilon - \alpha}{2\varepsilon}\frac{r}{n} + \frac{1}{\varepsilon}\frac{a}{n} \\ &\geq \frac{\varepsilon - \alpha}{2\varepsilon}\frac{k}{n}. \end{aligned}$$

Recalling $|S'| = (\varepsilon - \alpha)n/2$, we have:

$$P(N_v \geq k) \leq \left(1 - \frac{\varepsilon - \alpha}{2\varepsilon}\frac{k}{n}\right)^{(\varepsilon-\alpha)n/2} \leq \exp\left(-\frac{(\varepsilon - \alpha)^2 k}{4\varepsilon}\right).$$

Using this tail bound, we can bound the expectation of N_v:

$$EN_v = \sum_{k=0}^{\infty} P(N_v \geq k) \leq \sum_{k=0}^{\infty} \exp\left(-\frac{(\varepsilon - \alpha)^2 k}{4\varepsilon}\right) = O\left(\int_0^{\infty} \exp\left(-\frac{(\varepsilon - \alpha)^2 x}{4\varepsilon}\right) dx\right) = O\left(\frac{\varepsilon}{(\varepsilon - \alpha)^2}\right).$$

□

Composing Lemma 7.3.4 and Lemma 7.4.6, we can bound $\mathrm{cost}(V \setminus S')$. Let G' denote the subgraph of G induced by all vertices in $V \setminus S'$ that are not clustered by the pivots chosen in S'. Then we have the same three properties as in the proof of Theorem 7.4.1, which we repeat for convenience:

- Let T' be the set of all bad triangles in G'. We have $\sum_{t \in T} 1_{A_t \cap V \setminus S' \neq \emptyset} \geq |T'|$.

- The positive degree in G^0 of a vertex v is exactly N_v (as defined in Lemma 7.4.6.)
- Let E^0 denote the edge set of G^0 and E the disagreements made by Opt. Then Opt induces a clustering of G^0 that makes disagreements in $E^0 \setminus E$.

Using Lemma 7.3.4 and Lemma 7.4.6 and the above three properties, we conclude:

$$\text{cost}(V \cap S^0) \quad E|T^0| \quad E\Big[\sum_{ij \in E^0 \setminus E} d^+_{G^0}(i) + d^+_{G^0}(j)\Big] \quad \sum_{ij \in E} EN_i + EN_j = O\Big(\frac{e}{(e \quad a)^2}\Big)\text{Opt}:$$

7.5 Experiments

In this section, we empirically validate our theoretical findings. Specifically, we:

- Demonstrate that Pivot's performance can be poor when the arrival sequence is chosen adversarially.
- Verify that with a small portion of nodes randomly drawn from the dataset (which we refer to as *advice*), semi-online Pivot is competitive with offline Pivot in random order (which is a 3-approximation to optimum). Further, we show that the result is robust across multiple parameter settings and additional corruptions.
- Show that when the advice is temporal (and not random) the performance is still strong.

Datasets. We use the following datasets from the Stanford Large Network Dataset Collection [101][1]. Each dataset is a social network where the nodes represent entities such as users or communities, and the edges represent connections between the entities such as communication or transactions. We use two non-temporal datasets: ego-Facebook and ego-Gplus [108]; and two temporal datasets: soc-RedditHyperlinks [94] and soc-sign-bitcoin-otc [96, 95]. The temporal datasets have timestamps on the edges representing when the connection happened. We refer to these datasets as FACEBOOK, GPLUS, REDDIT and BITCOIN respectively. See Table 7.1 for a full description of the four data sources.

To generate signed complete graphs, on directed graphs we convert all arcs into undirected edges, and on weighted graphs we only keep the edges with positive weights. These edges represent the positive edges, and all the other edges are negative edges.

To perform a richer evaluation, we sub-sample from the datasets to obtain our input graphs in the following manner. The two non-temporal data sources, FACEBOOK and

[1] https://snap.stanford.edu/data/ego-Facebook.html, https://snap.stanford.edu/data/ego-Facebook.html, ego-Gplus https://snap.stanford.edu/data/ego-Gplus.html, soc-RedditHyperlinks https://snap.stanford.edu/data/soc-RedditHyperlinks.html, soc-sign-bitcoin-otc https://snap.stanford.edu/data/soc-sign-bitcoin-otc.html

Table 7.1 Dataset desciption.

Dataset	(# Edges, # Nodes)	Edge Features	Edge Meaning
FACEBOOK	(4039, 88234)	Undirected, unweighted, non-temporal	Users are friends
GPLUS	(107614, 13673453)	Directed, unweighted, non-temporal	One user follows another
REDDIT	(55863, 858490)	Directed, weighted, temporal	A hyperlink between two subreddits
BITCOIN	(5881, 35592)	Directed, weighted, temporal	A trust rating between users

GPLUS, contain ego-networks which represent the lists of friends of pre-selected users and the connections among them. We experiment on these ego-networks which are sub-graphs induced from the entire graph by a subset of nodes. For the temporal data sources, we take all edges with timestamps falling into a time interval and all nodes induced on these edges.

Performance Metrics. We use Pivot in uniformly random order as our benchmark. For each arrival order constructed, we conduct multiple trials, take the ratio of the average number of disagreements for semi-online Pivot over that of random Pivot, and subtract 1 to measure relative improvement, which we call degradation.

Power of Advice. Our first experiment studies the benefit of a random sample of nodes. We compare the performance of Pivot with and without advice under different element orderings.

We first test Pivot on a uniform random ordering of the nodes. Recall that this is a 3-approximation [18] to the optimum. Then we test Pivot in the following adversarial orders: (1) node degree in descending order; (2) number of bad triangles in a node's neighborhood that contains this node, in descending order[2]; and (3) timestamp, in chronological order (only for temporal datasets.[3])

Next, we explore the performance of semi-online Pivot with advice in the one-random sample model. We produce the advice by randomly sampling a fraction of the nodes. Then, in the online phase, the remaining nodes arrive in adversarial order. We test the same adversarial orders as before.

Table 7.2 Mean degradation and standard deviation in Pivot's performance on different sequences, $\varepsilon = 0.1$, 30 trials.

Dataset	(# Nodes, # Edges)	Random	Degree	Degree w. Advice	Bad Triangles	Bad Triangles w. Advice	Time	Time w. Advice
FACEBOOK	(534, 9626)	0 11.21 %	108.61 %	4.33 12.68 %	108.54 %	3.53 13.46 %	N/A	N/A
FACEBOOK	(1034, 53498)	0 6.00 %	63.19 %	0.19 5.49 %	63.72 %	1.48 6.63 %	N/A	N/A
GPLUS	(1650, 166292)	0 20.79 %	133.46 %	9.90 23.30 %	133.46 %	-0.80 17.03 %	N/A	N/A
GPLUS	(3455, 435569)	0 15.35 %	229.37 %	-3.95 16.03 %	229.38 %	-3.91 12.78 %	N/A	N/A
REDDIT	(4277, 9524)	0 42.79 %	610.64 %	3.99 8.31 %	610.97 %	5.71 8.49 %	139.26 %	-4.88 10.43%
REDDIT	(7019, 20724)	0 11.36 %	690.05 %	21.28 28.61 %	690.27 %	18.40 11.74 %	79.89%	4.59 8.65 %
REDDIT	(14042, 56567)	0 20.23 %	822.60 %	11.09 42.99 %	822.64 %	2.59 12.24 %	50.24 %	-1.01 14.11 %
BITCOIN	(2979, 14695)	0 21.83 %	1070.07 %	9.75 24.32 %	1066.81 %	11.70 25.48 %	579.06 %	8.50 40.54 %

[2] For a node with k neighbors, this number is equivalent to its clustering coefficient times $\frac{k(k-1)}{2}$.

[3] The original temporal datasets only have timestamps on edges. We consider the timestamp of any node to be the earliest timestamp of all edges adjacent to it.

See Table 7.2 for data on the performance of Pivot in different scenarios. Each entry shows the level of degradation compared with fully random Pivot after 30 trials (mean ± standard deviation). See Section C.1 for results on all input graphs. The first two columns show the data source and the size of the graph instance. Column "Random" shows random Pivot's performance. Columns "Degree", "Bad Triangles" and "Time" give Pivot's performance on the three adversarial orders mentioned earlier, while the columns next to these three show the performance of semi-online Pivot using the advice.

Compared with a fully random sequence, an adversarial sequence of online node arrivals can cause Pivot's performance to degrade significantly, from 2-10x. Sorting the nodes according to degrees or bad triangles in the neighborhood could causes Pivot's performance to be 1000% worse for some sub-samples. For the temporal node sequences, Pivot's performance is about 50 – 100% worse for REDDIT and 500% worse for BITCOIN. However, when $\varepsilon = 0.1$, semi-online Pivot's performance is only slightly worse (about 10 – 20%) than fully random Pivot. Thus, with only a small advice set, semi-online Pivot almost completely bridges the gap between adversarial and random order.

Robustness. Our next goal is to show that the gains of Pivot with advice are robust to different choices of ε as well as adversarial corruptions. Recall that for a node set of size n, the advice has $(\varepsilon - \alpha)n$ randomly sampled nodes and αn adversarially chosen nodes. For each of the four data sources, we select a sufficiently large sub-sample and change ε and α while producing advice.

For the corrupted advice, for any one of the three adversarial sequences, we choose the first α fraction of nodes to be the corruptions (i.e. the "worst" nodes as evidenced by the experiments with no advice). Then, we sample $(\varepsilon - \alpha)n$ nodes from the remaining nodes to complete the advice. The online phase arrival order is the same as in the adversarial sequence.

Figure 7.1(i) shows how degradation changes when ε ranges in $[0.01, 0.5]$ and $\alpha = 0$ (no corruptions), and (ii) shows how corruptions in the advice affect semi-online Pivot when $\varepsilon = 0.2$. Pivot's performance is robust against shrinking the size of the advice and against adding corruptions. For $\varepsilon \geq 0.05$, or with the advice set corrupted by 25%, the performance remains constant. See Section C.1 for results on more datasets.

Temporal advice. Finally, learning the advice from prior temporal data is sufficient for Pivot to have good performance. We test both temporal datasets: REDDIT and BITCOIN. The advice is generated from older data just prior to the time interval of the test dataset. In particular, our advice consists of all nodes in the older dataset, which are incident on edges

Fig. 7.1 (i) Degradation v.s. ϵ values, REDDIT, (# nodes, # edges) = (14042, 56567). (ii) Degradation v.s. α values, where $\epsilon = 0.2$.

in the test dataset (i.e. the old data that is related to the new data.) After the advice, the test data set arrives in adversarial order - including temporal order, where vertices arrive in order of their timestamps.

Table 7.3 Degradation of semi-online Pivot when using advice from historical data, REDDIT, 100 trials, test dataset has the duration of 6 months.

Days	e-value	Random	Degree	Degree w. Advice	Bad Triangles	Bad Triangles w. Advice	Time	Time w. Advice
5	0.09	0 14.57 %	675.40 %	60.73 38.29 %	675.12 %	65.73 46.47 %	52.08 %	55.49 29.57 %
10	0.15	0 14.57 %	675.40 %	42.09 25.19 %	675.12 %	36.85 20.26 %	52.08 %	34.74 23.39 %
15	0.19	0 14.57 %	675.40 %	41.09 32.35 %	675.12 %	36.85 20.26 %	52.08 %	36.58 37.41 %
20	0.22	0 14.57 %	675.40 %	30.76 23.27 %	675.12 %	36.43 32.97 %	52.08 %	41.66 48.21 %
25	0.25	0 14.57 %	675.40 %	35.28 50.72 %	675.12 %	30.97 32.24 %	52.08 %	30.54 28.82 %

Table 7.3 shows the results for REDDIT when we change the time interval used to generate the advice, which is shown in Column "Days" in time unit of days. The time interval for the test dataset is fixed to be 6 months. Column "e-value" gives the proportion of nodes in test dataset that appeared in the old data, which corresponds to the parameter ϵ in the ϵ-random sample model. See Section C.1 for the results for BITCOIN. The rest of the table shows the degradation of Pivot using the advice from the corresponding historical data. Pivot's performance improves when we increase the time interval (equivalent to increasing ϵ). Starting with $\epsilon = 0.15$ (time interval for old data/test data is 10 days/6 months), the temporal advice improves Pivot's performance on all adversarial orders. This demonstrates that Pivot works well empirically using advice from historical data.

7.6 Conclusion

By augmenting a standard online algorithm for correlation clustering with a small random sample of the data, we can overcome strong lower bounds for online correlation clustering. We give near-optimal algorithms for semi-online correlation clustering given a random

sample (with corruptions), and further this theory is predictive of practical performance. Empirically, with only a small random sample, semi-online pivot is competitive with the offline pivot in random order. Further, in temporal datasets, the sample can be practically obtained from past data. We show that semi-online models, where we augment an online algorithm with some of ine information, can be a powerful tool in both theory and practice to improve the performance of online algorithms, and we believe they will find further applications in other problems.

References

[big] https://cloud.google.com/bigquery-ml/docs/bigqueryml-intro.

[2] (2018). Kaggle machine learning and data science survey. https://www.kaggle.com/kaggle/kaggle-survey-2018.

[3] Abboud, A., Cohen-Addad, V., and Houdrougé, H. (2019). Subquadratic high-dimensional hierarchical clustering. In *Advances in Neural Information Processing Systems*, pages 11576–11586.

[4] Abo-Khamis, M., Im, S., Moseley, B., Pruhs, K., and Samadian, A. (2021a). Approximate aggregate queries under additive inequalities. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 85–99. SIAM.

[5] Abo-Khamis, M., Im, S., Moseley, B., Pruhs, K., and Samadian, A. (2021b). A relational gradient descent algorithm for support vector machine training. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 100–113. SIAM.

[6] Abo Khamis, M., Ngo, H. Q., Nguyen, X., Olteanu, D., and Schleich, M. (2018a). Ac/dc: in-database learning thunderstruck. In *Second Workshop on Data Management for End-To-End Machine Learning*, page 8. ACM.

[7] Abo Khamis, M., Ngo, H. Q., Nguyen, X., Olteanu, D., and Schleich, M. (2018b). In-database learning with sparse tensors. In *ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 325–340.

[8] Abo Khamis, M., Ngo, H. Q., and Rudra, A. (2016). Faq: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, page 13–28, New York, NY, USA. Association for Computing Machinery.

[9] Ackerman, M. and Ben-David, S. (2016). A characterization of linkage-based hierarchical clustering. *Journal of Machine Learning Research*, 17:232:1–232:17.

[10] Aggarwal, A., Deshpande, A., and Kannan, R. (2009). Adaptive sampling for k-means clustering. In *International Conference on Approximation Algorithms for Combinatorial Optimization Problems*, pages 15–28.

[11] Agrawal, R., Halverson, A., Kenthapadi, K., Mishra, N., and Tsaparas, P. (2009). Generating labels from clicks. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 172–181.

[12] Ahmadi, S., Galhotra, S., Saha, B., and Schwartz, R. (2020). Fair correlation clustering. arXiv:2002.03508

[13] 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*

[14] Ahmadian, S., Epasto, A., Knittel, M., Kumar, R., Mahdian, M., Moseley, B., Pham, P., Vassilvitskii, S., and Wang, Y. (2020b). Fair hierarchical clustering. In *Advances in Neural Information Processing Systems*, 33.

[15] Ahmadian, S., Epasto, A., Kumar, R., and Mahdian, M. (2019). Clustering without over-representation. In *KDD*, pages 267–275.

[16] Ahmadian, S., Epasto, A., Kumar, R., and Mahdian, M. (2020c). Fair correlation clustering. In *AISTATS*

[17] Ailon, N., Bhattacharya, A., and Jaiswal, R. (2018). Approximate correlation clustering using same-cluster queries. In *Latin American Symposium on Theoretical Informatics*, pages 14–27. Springer.

[18] Ailon, N., Charikar, M., and Newman, A. (2008). Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):1–27.

[19] Andoni, A. and Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122.

[20] Andoni, A., Nikolov, A., Onak, K., and Yaroslavtsev, G. (2014). Parallel algorithms for geometric graph problems. In *STOC*

[21] Arasu, A., Ré, C., and Suciu, D. (2009). Large-scale deduplication with constraints using dedupalog. In *2009 IEEE 25th International Conference on Data Engineering*, pages 952–963. IEEE.

[22] Arthur, D. and Vassilvitskii, S. (2007). k-means++: the advantages of careful seeding. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035.

[23] Atserias, A., Grohe, M., and Marx, D. (2008). Size bounds and query plans for relational joins. In *IEEE Symposium on Foundations of Computer Science*, pages 739–748.

[24] Awasthi, P., Blum, A., and Sheffet, O. (2012). Center-based clustering under perturbation stability. *Inf. Process. Lett.*, 112(1-2):49–54.

[25] Backurs, A., Indyk, P., Onak, K., Schieber, B., Vakilian, A., and Wagner, T. (2019). Scalable fair clustering. In *ICML*, pages 405–413.

[26] Bahmani, B., Moseley, B., Vattani, A., Kumar, R., and Vassilvitskii, S. (2012). Scalable k-means++. *PVLDB*, 5(7):622–633.

[27] Balcan, M., Blum, A., and Gupta, A. (2013). Clustering under approximation stability. *J. ACM*, 60(2):8:1–8:34.

[28] Balcan, M., Blum, A., and Vempala, S. (2008). A discriminative framework for clustering via similarity functions. In Proceedings of STOC, pages 671–680.

[29] Bansal, N., Blum, A., and Chawla, S. (2004). Correlation clustering. Machine learning, 56(1):89–113.

[30] Barocas, S., Hardt, M., and Narayanan, A. (2019). Fairness and Machine Learning. www.fairmlbook.org.

[31] Bateni, M., Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Kiveris, R., Lattanzi, S., and Mirrokni, V. S. (2017). Affinity clustering: Hierarchical clustering at scale. In NeurIPS.

[32] Bera, S., Chakrabarty, D., Flores, N., and Negahbani, M. (2019). Fair algorithms for clustering. In NeurIPS, pages 4955–4966.

[33] 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.

[34] Bonchi, F., Gionis, A., and Ukkonen, A. (2013). Overlapping correlation clustering. Knowledge and information systems, 35(1):1–32.

[35] Borodin, A., Ostrovsky, R., and Rabani, Y. (1999). Subquadratic approximation algorithms for clustering problems in high dimensional spaces. In Proceedings of the thirty- rst annual ACM symposium on Theory of computing, pages 435–444.

[36] Braverman, V., Frahling, G., Lang, H., Sohler, C., and Yang, L. F. (2017). Clustering high dimensional dynamic data streams. In International Conference on Machine Learning, pages 576–585.

[37] Calinescu, G., Chekuri, C., PÃ¡l, M., and VondrÃ¡k, J. (2011). Maximizing a monotone submodular function subject to a matroid constraint. SIAM Journal on Computing, 40(6):1740–1766.

[38] Carlsson, G. E. and Mémoli, F. (2010). Characterization, stability and convergence of hierarchical clustering methods. Journal of Machine Learning Research, 11:1425–1470.

[39] Casella, G., Robert, C. P., Wells, M. T., et al. (2004). Generalized accept-reject sampling schemes. In A Festschrift for Herman Rubin, pages 342–347. Institute of Mathematical Statistics.

[40] Celis, L. E., Huang, L., and Vishnoi, N. K. (2018a). Multiwinner voting with fairness constraints. In IJCAI, pages 144–151.

[41] Celis, L. E., Straszak, D., and Vishnoi, N. K. (2018b). Ranking with fairness constraints. In ICALP, pages 28:1–28:15.

[42] Chakrabarti, D., Kumar, R., and Punera, K. (2008). A graph-theoretic approach to webpage segmentation. In Proceedings of the 17th international conference on World Wide Web, pages 377–386.

[43] Charikar, M. and Chatziafratis, V. (2017). Approximate hierarchical clustering via sparsest cut and spreading metrics. In SODA, pages 841–854.

[44] Charikar, M., Chatziafratis, V., and Niazadeh, R. (2019a). Hierarchical clustering better than average-linkage. In SODA, pages 2291–2304.

[45] Charikar, M., Chatziafratis, V., Niazadeh, R., and Yaroslavtsev, G. (2019b). Hierarchical clustering for euclidean data. In Chaudhuri, K. and Sugiyama, M., editors, AISTATS, volume 89, pages 2721–2730. PMLR.

[46] Chatziafratis, V., Niazadeh, R., and Charikar, M. (2018). Hierarchical clustering with structural constraints. In Proceedings of the ICML, pages 773–782.

[47] Chawla, S., Makarychev, K., Schramm, T., and Yaroslavtsev, G. (2015). Near optimal lp rounding algorithm for correlationclustering on complete and complete k-partite graphs. In Proceedings of the forty-seventh annual ACM symposium on Theory of computing, pages 219–228.

[48] Chen, X., Fain, B., Lyu, C., and Munagala, K. (2019). Proportionally fair clustering. In ICML, pages 1032–1041.

[49] Chen, Y., Sanghavi, S., and Xu, H. (2012). Clustering sparse graphs. In Proceedings of the 25th International Conference on Neural Information Processing Systems-Volume 2, pages 2204–2212.

[50] Cheng, Z. and Koudas, N. (2019). Nonlinear models over normalized data. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1574–1577. IEEE.

[51] Chhabra, A. and Mohapatra, P. (2020). Fair algorithms for hierarchical agglomerative clustering. arXiv:2005.03197.

[52] Chierichetti, F., Kumar, R., Lattanzi, S., and Vassilvitskii, S. (2017). Fair clustering through fairlets. In NIPS, pages 5029–5037.

[53] Chierichetti, F., Kumar, R., Lattanzi, S., and Vassilvitskii, S. (2019). Matroids, matchings, and fairness. In AISTATS, pages 2212–2220.

[54] Chierichetti, F., Kumar, R., and Mahdian, M. (2014). The complexity of LSH feasibility. Theor. Comput. Sci., 530:89–101.

[55] Chiplunkar, A., Kale, S., and Ramamoorthy, S. N. (2020). How to solve fair k-center in massive data models. In ICML.

[56] Cochez, M. and Neri, F. (2015). Scalable hierarchical clustering: Twister tries with a posteriori trie elimination. In SSCI, pages 756–763.

[57] Cohen-Addad, V., Hjuler, N., Parotsidis, N., Saulpic, D., and Schwiegelshohn, C. (2019). Fully dynamic consistent facility location. In NeurIPS'19-33rd Conference on Neural Information Processing Systems.

[58] Cohen-Addad, V., Kanade, V., Mallmann-Trenn, F., and Mathieu, C. (2018). Hierarchical clustering: Objective functions and algorithms. In SODA, pages 378–397.

[59] Cohen-addad, V., Kanade, V., Mallmann-trenn, F., and Mathieu, C. (2019). Hierarchical clustering: Objective functions and algorithms. J. ACM, 66(4).

[60] Curtin, R. R., Moseley, B., Ngo, H. Q., Nguyen, X., Olteanu, D., and Schleich, M. (2020). Rk-means: Fast clustering for relational data. In Chiappa, S. and Calandra, R., editors, The 23rd International Conference on Arti cial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy], volume 108 of Proceedings of Machine Learning Research, pages 2742–2752. PMLR.

[61] Dasgupta, S. (2016). A cost function for similarity-based hierarchical clustering. In STOC

[62] Dasgupta, S. and Gupta, A. (2003). An elementary proof of a theorem of johnson and lindenstrauss. Random Struct. Algorithms, 22(1):60–65.

[63] Dasgupta, S. and Long, P. M. (2005). Performance guarantees for hierarchical clustering. J. Comput. Syst. Sci., 70(4):555–569.

[64] Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In Proceedings of the twentieth annual symposium on Computational geometry, pages 253–262.

[65] Demaine, E. D., Emanuel, D., Fiat, A., and Immorlica, N. (2006). Correlation clustering in general weighted graphs. Theoretical Computer Science, 361(2-3):172–187.

[66] Ene, A., Im, S., and Moseley, B. (2011). Fast clustering using MapReduce. In KDD, pages 681–689.

[67] Garey, M. R. and Johnson, D. S. (2002). Computers and Intractability. WH Freeman New York.

[68] Ghoshdastidar, D., Perrot, M., and von Luxburg, U. (2018). Foundations of comparison-based hierarchical clustering. CoRR, abs/1811.00928.

[69] Giotis, I. and Guruswami, V. (2005). Correlation clustering with a xed number of clusters. arXiv preprint cs/0504023.

[70] Goodman, J. E. and O'Rourke, J., editors (1997). Handbook of Discrete and Computational Geometry. CRC Press, Inc., USA.

[Google Cloud Platform] Google Cloud Platform. https://cloud.google.com/.

[72] Guha, S., Meyerson, A., Mishra, N., Motwani, R., and O'Callaghan, L. (2003). Clustering data streams: Theory and practice. IEEE Transactions on Knowledge and Data Engineering, 15(3):515–528.

[73] Guha, S., Rastogi, R., and Shim, K. (2001). Cure: An ef cient clustering algorithm for large databases. Inf. Syst., 26(1):35–58.

[74] Guo, X., Kulkarni, J., Li, S., and Xian, J. (2020). On the facility location problem in online and dynamic models. In Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[75] Guo, X., Kulkarni, J., Li, S., and Xian, J. (2021). Consistent k-median: Simpler, better and robust. In International Conference on Arti cial Intelligence and Statistics, pages 1135–1143. PMLR.

[76] Huang, L., Jiang, S. H. C., and Vishnoi, N. K. (2019). Coresets for clustering with fairness constraints. In NeurIPS, pages 7587–7598.

[77] Indyk, P. (2001). Algorithmic applications of low-distortion geometric embeddings. In FOCS.

[78] Indyk, P. and Matousek, J. (2004). Low-distortion embeddings of nite metric spaces. In in Handbook of Discrete and Computational Geometry, pages 177–196. CRC Press.

[79] Indyk, P. and Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In Proceedings of the thirtieth annual ACM symposium on Theory of computing, pages 604–613.

[80] Jin, C., Chen, Z., Hendrix, W., Agrawal, A., and Choudhary, A. N. (2015a). Incremental, distributed single-linkage hierarchical clustering algorithm using mapreduce. In HPC.

[81] Jin, C., Liu, R., Chen, Z., Hendrix, W., Agrawal, A., and Choudhary, A. N. (2015b). A scalable hierarchical clustering algorithm using spark. In Big Data Computing Service and Applications.

[82] Jones, M., Nguyen, T., and Nguyen, H. (2020). Fair k-centers via maximum matching. In ICML.

[83] Kalashnikov, D. V., Chen, Z., Mehrotra, S., and Nuray-Turan, R. (2008). Web people search via connection analysis. IEEE Transactions on Knowledge and Data Engineering, 20(11):1550–1565.

[84] Kanungo, T., Mount, D. M., Netanyahu, N. S., Piatko, C. D., Silverman, R., and Wu, A. Y. (2004). A local search approximation algorithm for k-means clustering. Computational Geometry, 28(2-3):89–112.

[85] Karloff, H. J., Suri, S., and Vassilvitskii, S. (2010). A model of computation for mapreduce. In SODA.

[86] Khamis, M. A., Ngo, H. Q., Ré, C., and Rudra, A. (2016). Joins via geometric resolutions: Worst case and beyond. ACM Transactions on Database Systems, 41(4):22:1–22:45.

[87] Kleinberg, J., Lakkaraju, H., Leskovec, J., Ludwig, J., and Mullainathan, S. (2017). Human decisions and machine predictions. The Quarterly Journal of Economics, 133(1):237–293.

[88] Kleindessner, M., Awasthi, P., and Morgenstern, J. (2019a). k-center clustering for data summarization. In ICML, pages 3448–3457.

[89] Kleindessner, M., Samadi, S., Awasthi, P., and Morgenstern, J. (2019b). Guarantees for spectral clustering with fairness constraints. In ICML, pages 3448–3457.

[90] Korula, N., Mirrokni, V., and Zadimoghaddam, M. (2018). Online submodular welfare maximization: Greedy beats 1/2 in random order. SIAM Journal on Computing, 47(3):1056–1086.

[91] Kull, M. and Vilo, J. (2008). Fast approximate hierarchical clustering using similarity heuristics. BioData Mining, 1.

[92] Kumar, A., Naughton, J., and Patel, J. M. (2015). Learning generalized linear models over normalized data. In ACM SIGMOD International Conference on Management of Data, pages 1969–1984.

[93] Kumar, R., Purohit, M., Schild, A., Svitkina, Z., and Vee, E. (2019). Semi-online bipartite matching. In 10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA, pages 50:1–50:20.

[94] Kumar, S., Hamilton, W. L., Leskovec, J., and Jurafsky, D. (2018a). Community interaction and con ict on the web. In Proceedings of the 2018 world wide web conference, pages 933–943.

[95] Kumar, S., Hooi, B., Makhija, D., Kumar, M., Faloutsos, C., and Subrahmanian, V. (2018b). Rev2: Fraudulent user prediction in rating platforms. In Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, pages 333–341.

[96] Kumar, S., Spezzano, F., Subrahmanian, V., and Faloutsos, C. (2016). Edge weight prediction in weighted signed networks. In 2016 IEEE 16th International Conference on Data Mining (ICDM), pages 221–230. IEEE.

[97] Lattanzi, S., Lavastida, T., Lu, K., and Moseley, B. (2019). A framework for parallelizing hierarchical clustering methods. In ECML.

[98] Lattanzi, S., Moseley, B., Vassilvitskii, S., Wang, Y., and Zhou, R. (2021). Robust online correlation clustering. Advances in Neural Information Processing Systems, 34.

[99] Lattanzi, S. and Vassilvitskii, S. (2017). Consistent k-clustering. In International Conference on Machine Learning, pages 1975–1984. PMLR.

[100] Lavastida, T., Lu, K., Moseley, B., and Wang, Y. (2021). Scaling average-linkage via sparse cluster embeddings. In Asian Conference on Machine Learning, pages 1429–1444. PMLR.

[101] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[102] Li, S. and Svensson, O. (2016). Approximating k-median via pseudo-approximation. SIAM J. Comput., 45(2):530–547.

[103] Liberty, E., Sriharsha, R., and Sviridenko, M. (2016). An algorithm for online k-means clustering. In 2016 Proceedings of the eighteenth workshop on algorithm engineering and experiments (ALENEX), pages 81–89. SIAM.

[104] Lichman, M. (2013). UCI ml repository.

[105] Lykouris, T., Mirrokni, V., and Paes Leme, R. (2018). Stochastic bandits robust to adversarial corruptions. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 114–122.

[106] Lykouris, T. and Vassilvitskii, S. (2018). Competitive caching with machine learned advice. In *International Conference on Machine Learning*, pages 3296–3305. PMLR.

[107] Mathieu, C., Sankur, O., and Schudy, W. (2010). Online correlation clustering. In *27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010, March 4-6, 2010, Nancy, France*, pages 573–584.

[108] McAuley, J. J. and Leskovec, J. (2012). Learning to discover social circles in ego networks. In *NIPS*, volume 2012, pages 548–56. Citeseer.

[109] Menon, A. K., Rajagopalan, A., Sumengen, B., Citovsky, G., Cao, Q., and Kumar, S. (2019). Online hierarchical clustering approximations. *CoRR*, abs/1909.09667.

[110] Meyerson, A. (2001). Online facility location. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 426–431. IEEE.

[111] Meyerson, A., O'Callaghan, L., and Plotkin, S. A. (2004). A k-median algorithm with running time independent of data size. *Machine Learning*, 56(1-3):61–87.

[112] Monath, N., Kobren, A., Krishnamurthy, A., Glass, M. R., and McCallum, A. (2019). Scalable hierarchical clustering with tree grafting. In *SIGKDD*.

[113] Moseley, B., Pruhs, K., Samadian, A., and Wang, Y. (2021a). Relational algorithms for k-means clustering. In *International Colloquium on Automata, Languages, and Programming (ICALP)*.

[114] Moseley, B., Vassilvitskii, S., and Wang, Y. (2021b). Hierarchical clustering in general metric spaces using approximate nearest neighbors. In *International Conference on Arti cial Intelligence and Statistics*, pages 2440–2448. PMLR.

[115] Moseley, B. and Wang, J. (2017). Approximation bounds for hierarchical clustering: Average linkage, bisecting k-means, and local search. *Advances in neural information processing systems*, 30.

[116] Mullner, D. (2013). fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *Journal of Statistical Software*, 53(9).

[117] Murtagh, F. (1983). A survey of recent advances in hierarchical clustering algorithms. *Comput. J.*, 26(4):354–359.

[118] Murtagh, F. and Contreras, P. (2012). Algorithms for hierarchical clustering: an overview. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discov.*, 2(1):86–97.

[119] Ostrovsky, R., Rabani, Y., Schulman, L. J., and Swamy, C. (2012). The effectiveness of lloyd-type methods for the k-means problem. *Journal of the ACM (JACM)*, 59(6):28.

[120] Plaxton, C. G. (2006). Approximation algorithms for hierarchical location problems. *J. Comput. Syst. Sci.*, 72(3):425–443.

[121] Rendle, S. (2013). Scaling factorization machines to relational data. In Proceedings of the VLDB Endowment, volume 6, pages 337–348. VLDB Endowment.

[122] Rösner, C. and Schmidt, M. (2018). Privacy preserving clustering with constraints. In ICALP, pages 96:1–96:14.

[123] Roughgarden, T. (2020). Beyond the Worst-Case Analysis of Algorithms. Cambridge University Press.

[124] Roy, A. and Pokutta, S. (2017). Hierarchical clustering via spreading metrics. JMLR, 18:88:1–88:35.

[125] Saha, B. and Subramanian, S. (2019). Correlation clustering with same-cluster queries bounded by optimal cost. arXiv preprint arXiv:1908.04976.

[126] Schleich, M., Olteanu, D., and Ciucanu, R. (2016). Learning linear regression models over factorized joins. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pages 3–18. ACM.

[127] 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.

[128] Sohler, C. and Woodruff, D. P. (2018). Strong coresets for k-median and subspace approximation: Goodbye dimension. In Symposium on Foundations of Computer Science, pages 802–813.

[129] Steinbach, M., Karypis, G., Kumar, V., et al. (2000). A comparison of document clustering techniques. In KDD workshop on text mining, volume 400, pages 525–526. Boston.

[130] Vidal, R. (2006). Online clustering of moving hyperplanes. Advances in Neural Information Processing Systems, 19:1433–1440.

[131] Wang, D. and Wang, Y. (2018). An improved cost function for hierarchical cluster trees. CoRR, abs/1812.02715.

[132] Wang, Y. and Moseley, B. (2020). An objective for hierarchical clustering in euclidean space and its connection to bisecting k-means. In AAAI, pages 6307–6314.

[133] Xu, R. and Wunsch, D. C. (2005). Survey of clustering algorithms. IEEE Trans. Neural Networks, 16(3):645–678.

[Yang et al.] Yang, K., Gao, Y., Liang, L., Yao, B., Wen, S., and Chen, G. Towards factorized svm with gaussian kernels over normalized data.

[135] Yaroslavtsev, G. and Vadapalli, A. (2018). Massively parallel algorithms and hardness for single-linkage clustering under lp distances. In ICML.

[136] Yu, C. T. and Ozsoyoglu, M. Z. (1979). An algorithm for tree-query membership of a distributed query. In Computer Software and The IEEE Computer Society's Third International Applications Conference, pages 306–312. IEEE.

[137] Zhang, J., Ghahramani, Z., and Yang, Y. (2004). A probabilistic model for online document clustering with application to novelty detection. In *Advances in neural information processing systems 17*, pages 1617–1624.

[138] Zhao, Y., Karypis, G., and Fayyad, U. M. (2005). Hierarchical clustering algorithms for document datasets. *Data Min. Knowl. Discov.*, 10(2):141–168.

Appendix A

Background Information for Relational Database

A.1 Uniform Sampling From a Hypersphere

In order to uniformly sample a point from inside a ball, it is enough to show how we can count the number of points located inside a ball grouped by a table. Because, if we can count the number of points grouped by input tables, then we can use similar technique to the one used in Section 5.4 to sample. Unfortunately, as we discussed in Section 5.2, it is #P-Hard to count the number of points inside a ball; however, it is possible to obtain a approximation of the number of points [4]. Bellow we brie y explain the algorithm in [4] for counting the number of points inside a hypersphere.

Given a center c and a radius R, the goal is approximating the number of tuples $x \in J$ for which $\sum_i (c^i - x^i)^2 \leq R$. Consider the set S containing all the multisets of real numbers. We denote a multiset A by a set of pairs of $(v, f_A(v))$ where v is a real value and $f(v)$ is the frequency of v in A. For example, $A = \{(2.3, 10), (3.5, 1)\}$ is a multiset that has 10 members with value 2.3 and 1 member with value 3.5. Then, let $\uplus$ be the summation operator meaning $C = A \uplus B$ if and only if for all $x \in R$, $f_C(x) = f_A(x) + f_B(x)$, and let $\otimes$ be the convolution operator such that $C = A \otimes B$ if and only if $f_C(x) = \sum_{i \in R} f_A(i) + f_B(x - i)$. Then the claim is $(S, \uplus, \otimes)$ is a commutative semiring and the following SumProd query returns a multiset that has all the squared distances of the points in J from C:

$$\bigotimes_{x \in J} \bigotimes_i \{((x^i - c^i)^2, 1)\}$$

Using the result of the multiset, it is possible to count exactly the number of tuples $x \in J$ for which $\|x - c\|_2^2 \leq R^2$. However, the size of the result is as large as $\Omega(|J|)$.

In order to make the size of the partial results and time complexity of π and σ operators polynomial, the algorithm uses $(1+\delta)$ geometric bucketing. The algorithm returns an array where in j-th entry it has the smallest value r for which there are $(1+\delta)^j$ tuples $x \in J$ satisfying $\|x - c\|_2^2 \leq r^2$.

The query can also be executed grouped by one of the input tables. Therefore, using this polynomial approximation scheme, we can calculate conditioned marginalized probability distribution with multiplicative $(1 \pm \delta)$. Therefore, using m queries, it is possible to sample a tuple from a ball with probability distribution $\frac{1}{n}(1 \pm m\delta)$ where n is the number of points inside the ball. In order to get a sample with probability $\frac{1}{n}(1 \pm \varepsilon)$, all we need is to set $\delta = \varepsilon/m$; hence, on [4], the time complexity for sampling each tuple will be $\frac{m^9 \log^4(n)}{\varepsilon^2} Y(n, d, m)$

A.2 Background Information About Database Concepts

Given a tuple x, define $P_F(x)$ to be projection of x onto the set of features F. meaning $P_F(x)$ is a tuple formed by keeping the entries of x that are corresponding to the feature F in F. For example let T be a table with columns (A, B, C) and let $x = (1, 2, 3)$ be a tuple of T, then $P_{\{A,C\}}(x) = (1, 3)$.

Definition A.2.1 (Join). Let $T_1, \ldots, T_m$ be a set of tables with corresponding sets of columns/features $F_1, \ldots, F_m$ we define the join of them $J = T_1 \bowtie \cdots \bowtie T_m$ as a table such that the set of columns of J is $\bigcup_i F_i$, and $x \in J$ if and only if $P_{F_i}(x) \in T_i$.

Note that the above definition of join is consistent with the definition written in Section 5.1 but offers more intuition about what the join operation means geometrically.

Definition A.2.2 (Join Hypergraph). Given a join $J = T_1 \bowtie \cdots \bowtie T_m$, the hypergraph associated with the join is $H = (V, E)$ where V is the set of vertices and for every column in J there is a vertex x_i in V, and for every table T_i there is a hyper-edge e in E that has the vertices associated with the columns of T_i.

Theorem A.2.1 (AGM Bound [23]). Given a join $J = T_1 \bowtie \cdots \bowtie T_m$ with d columns and its associated hypergraph $H = (V, E)$, and let C be a subset of $\mathrm{col}(J)$, let $X = (x_1, \ldots, x_m)$ be

any feasible solution to the following Linear Programming:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} \log(|T_j|)\, x_j \\ \text{subject to} \quad & \sum_{j : v \in e_j} x_j \geq 1, \quad v \in C \\ & 0 \leq x_j \leq 1, \qquad j = 1, \ldots, t \end{aligned}$$

Then $\prod_i |T_i|^{x_i}$ is an upper bound for the cardinality of $F_C(J)$, this upperbound is tight if X is the optimal answer.

We give another definition of acyclicity which is consistent with the definition in the main body.

Definition A.2.3 (Acyclic Join). We call a join query (or a relational database schema) acyclic if one can repeatedly apply one of the two operations and convert the set of tables to an empty set:

1. Remove a column that is only in one table.
2. Remove a table for which its columns are fully contained in another table.

Definition A.2.4 (Hypertree Decomposition). Let $H = (V, E)$ be a hypergraph and $T = (V', E')$ be a tree with a subset of V associated to each vertex $v' \in V'$ called bag of v' and show it by $b(v')$ V. T is called a hypertree decomposition of H if the following holds:

1. For each hyperedge $e \in E$ there exists $v' \in V'$ such that e $b(v')$
2. For each vertex $v \in E$ the set of vertices in V' that have v in their bag are all connected in T.

Definition A.2.5. Let $H = (V, E)$ be a join hypergraph and $T = (V', E')$ be its hypertree decomposition. For each $v' \in V'$, let $X^{v'} = (x_1^{v'}, x_2^{v'}, \ldots, x_m^{v'})$ be the optimal solution to the following linear program: min $\sum_{j=1}^{t} x_j$, subject to $\sum_{j : v_i \in e_j} x_j$ $1, \forall v_i \in b(v')$ where 0 x_j 1 for each $j \in [t]$. Then the width of v' is $\sum_i x_i^{v'}$ denoted by $w(v')$ and the fractional width of T is $\max_{v' \in V'} w(v')$.

Definition A.2.6 (fhtw). Given a join hypergraph $H = (V, E)$, the fractional hypertree width of H, denoted by fhtw, is the minimum fractional width of its hypertree decomposition. Here the minimum is taken over all possible hypertree decompositions.

Observation 1. The fractional hypertree width of an acyclic join is 1, and each bag in its hypertree decomposition is a subset of the columns in some input table.

Theorem A.2.2 (Inside-out [36]). There exists an algorithm to evaluate a SumProd query in time $O(T m d^2 n^{fhtw} \log(n))$ where $fhtw$ is the fractional hypertree width of the query and T is the time needed to evaluate $\otimes$ and $\oplus$ for two operands. The same algorithm with the same time complexity can be used to evaluate SumProd queries grouped by one of the input tables.

Theorem A.2.3. Let Q_f be a function from domain of column f in J to $\mathbb{R}$, and G be a vector that has a row for each tuple $x \in T_i$. Then the query

$$\sum_{x \in J} \prod_{f} Q_f(x_f)$$

can be converted to a SumProd and the query returning G with definition

$$G_r = \sum_{x \in Y_{ij} \in J} \prod_{f} F_i(x_f)$$

can be converted to a SumProd query grouped by T_i

Proof. Let $S = \{(a,b) \mid a \in \mathbb{R}, b \in I\}$, and for any two pairs of $(a,b),(c,d) \in S$ we define:

$$(a,b) \oplus (c,d) = (a+c, b+d)$$

and

$$(a,b) \otimes (c,d) = (ad+cb, bd).$$

Then the theorem can be proven by using the following two claims:

1. $(S, \oplus, \otimes)$ forms a commutative semiring with identity zero $I_0 = (0,0)$ and identity one $I_1 = (0,1)$.
2. The query $\sum_{x \in J} \prod_{f} (Q_f(x_f), 1)$ is a SumProd FAQ where the first entry of the result is $\sum_{x \in J} \prod_{f} Q_f(x_f)$ and the second entry is the number of rows in J.

proof of the first claim: Since arithmetic summation is commutative and associative, it is easy to see $\oplus$ is also commutative and associative. Furthermore, based on the definition of $\otimes$ we have $(a,b) \otimes I_0 = (a+0, b+0) = (a,b)$.

The operator is also commutative since arithmetic multiplication is commutative, the associativity of can be proved by

$$\begin{aligned}(a_1;b_1) \quad ((a_2;b_2) \quad (a_3;b_3)) &= (a_1;b_1) \quad (a_2b_3 + a_3b_2;b_2b_3)\\ &= (a_1b_2b_3 + b_1a_2b_3 + b_1b_2a_3;b_1b_2b_3)\\ &= (a_1b_2 + b_1a_2;b_1b_2) \quad (a_3;b_3)\\ &= ((a_1;b_1) \quad (a_2;b_2)) \quad (a_3;b_3)\end{aligned}$$

Also note that based on the de nition of, $(a;b) \quad I_0 = I_0$ and $(a;b) \quad I_1 = (a;b)$. The only remaining property that we need to prove is the distribution of over :

$$\begin{aligned}(a;b) \quad ((c_1;d_1) \quad (c_2;d_2)) &= (a;b) \quad (c_1 + c_2;d_1 + d_2)\\ &= (a;b) \quad (c_1 + c_2;d_1 + d_2)\\ &= (c_1b + c_2b + ad_1 + ad_2;bd_1 + bd_2)\\ &= (c_1b + ad_1;bd_1) \quad (c_2b + ad_2;bd_2)\\ &= ((a;b) \quad (c_1;d_1)) \quad ((a;b) \quad (c_2;d_2))\end{aligned}$$

Now we can prove the second claim: To prove the second claim, since we have already shown the semiring properties of $(\mathbb{S}; \ ; \ )$ we only need to show what is the result of $\bigoplus_{x \in J} f(Q_f(x_f);1)$. We have $\bigoplus_f(Q_i(x_f);1) = (\sum_f Q_i(x_f);1)$, therefore

$$\bigoplus_{x \in J} f(Q_i(x_f);1) = \bigoplus_{x \in J}(\sum_f Q_f(x_f);1) = (\sum_{x \in J} \sum_f Q_f(x_f); \sum_{x \in J} 1)$$

where the rst entry is the result of the SumSum query and the second entry is the number of rows in J. □

Appendix B

Omitted Experimental Results for Hierarchical Clustering in General Metric Space

This chapter contains the experiment results omitted in the main body. We show the performance table and running time plots for Proxy-Hash-SL And Proxy-Hash-AL on the three road map datasets and seizure . These results lead to the same conclusions as in the main body: both Proxy-Hash-SL and Proxy-Hash-AL are more accurate and ef cient than directly using the proxy metric.

B.1 Performance And Running Time for Proxy-Hash-SL And Proxy-Hash-AL

We defer the performance data for Proxy-Hash-SL on road map dataset in Bay Area and Great Lakes until the next subsection, where we will show the complete data table containing performance data for a range of values in the LSH algorithm, gathered from the road map datasets in all three cities. Here we show the performance for Proxy-Hash-AL on road maps in Bay Area and Great Lakes in Table B.1 and B.2.

Running-time wise, Figure B.1a and B.1b show the growth in number of distance computations in Proxy-Hash-AL for road map in Bay Area and the Great Lakes, versus growth in sample size. The sample sizes are plotted on log scale. Both curves are strictly sub-quadratic as it is dominated by $y = cx^{1.3}$ where c is a constant.

Table B.1 Comparing the performance of different average linkage methods, Bay Area

Sample Size	299	440	481	830	909	1065	1280	1363
Aprx Ratio, mean, Proxy-AL	3.608	2.491	3.423	2.830	2.431	3.017	2.787	2.175
Aprx Ratio, mean, Proxy-Hash-AL	1.484	1.647	1.493	1.725	1.629	1.709	1.821	1.730
Aprx Ratio, 90%, Proxy-AL	6.554	4.327	5.324	3.759	3.816	5.156	4.711	3.418
Aprx Ratio, 90%, Proxy-Hash-AL	2.024	2.300	1.987	2.402	2.214	2.313	2.550	2.353
Aprx Ratio, max, Proxy-AL	85.784	126.459	89.782	116.732	44.144	96.401	215.839	64.011
Aprx Ratio, max, Proxy-Hash-AL	2.553	3.223	3.184	3.759	3.816	3.646	4.507	3.418
Global Obj, Proxy-AL	0.994	0.990	0.990	0.998	0.999	0.994	0.963	1.001
Global Obj, Proxy-Hash-AL	1.004	0.970	0.985	0.975	0.999	0.987	0.966	0.971

Table B.2 Comparing the performance of different average linkage methods, the Great Lakes

Sample Size	86	211	363	496	632	713	1064	1885
Aprx Ratio, mean, Proxy-AL	1.539	1.457	1.935	1.971	1.975	1.742	1.828	1.837
Aprx Ratio, mean, Proxy-Hash-AL	1.362	1.545	1.562	1.592	1.603	1.675	1.766	1.797
Aprx Ratio, 90%, Proxy-AL	2.158	1.732	2.476	2.516	3.280	2.839	2.466	2.658
Aprx Ratio, 90%, Proxy-Hash-AL	1.890	2.311	2.112	2.172	2.158	2.268	2.523	2.412
Aprx Ratio, max, Proxy-AL	14.361	8.603	55.200	45.255	28.929	17.767	65.448	51.816
Aprx Ratio, max, Proxy-Hash-AL	2.428	3.367	3.043	2.971	3.079	3.137	5.073	3.943
Global Obj, Proxy-AL	0.992	0.996	0.990	1.001	0.988	0.997	0.990	0.996
Global Obj, Proxy-Hash-AL	0.983	0.998	0.996	0.983	0.994	0.995	0.992	0.997

Table B.3 Ratio between total road distance of the tree and real MST, New York

Sample Size	169	330	727	1166	1825	3765	6710	14428	28985
Euclidean	1.760	1.304	1.554	1.784	1.412	1.805	1.753	1.883	1.511
Proxy-Hash-SL-0.1	1.035	1.016	1.021	1.024	1.027	1.030	1.029	1.022	1.024
Proxy-Hash-SL-0.2	1.048	1.035	1.055	1.038	1.045	1.040	1.066	1.044	1.055
Proxy-Hash-SL-0.3	1.109	1.030	1.055	1.052	1.062	1.060	1.063	1.052	1.059
Proxy-Hash-SL-0.4	1.082	1.036	1.075	1.064	1.068	1.064	1.063	1.074	1.078
Proxy-Hash-SL-0.5	1.074	1.094	1.081	1.091	1.065	1.068	1.090	1.072	1.077

In the main body, we have also omitted the running time plots for Proxy-Hash-SL for all data sets. See Figure B.2 for them. Clearly, all running time curves are strictly subquadratic, and even only slightly super-linear on all road map datasets.

B.1.1 Robustness of Performance Against Sample Sizes And Parameter Tuning

There are a lot of parameters in the implementation of both Proxy-Hash-SL and Proxy-Hash-AL, which might affect the accuracy and ef ciency of our algorithms. One of the most important parameters is- in every round of LSH the threshold merging value grows by a factor of $1+ e$. In this section we show that the performance of our algorithms are robust against differen e values and the growth of sample sizes. We focus on Proxy-Hash-SL for now.

See Table 6.3, B.4, B.5 for partial statistics of Proxy-Hash-SL performance for Bay Area and the Great Lakes. The sample sizes are picked to grow by approximately a fa tor of 2. Here

(a) Bay Area (b) the Great Lakes

Fig. B.1 Growth of distance computation, Proxy-Hash-AL, Bay Area and Great Lakes

Table B.4 Ratio between total road distance of the tree and real MST, Bay Area

Sample Size	251	424	909	1804	3732	7135	10106	22861
Euclidean	2.241	1.643	1.954	1.407	1.905	2.252	2.198	2.397
Proxy-Hash-SL-0.1	1.027	1.039	1.032	1.026	1.035	1.026	1.028	1.027
Proxy-Hash-SL-0.2	1.058	1.045	1.038	1.046	1.058	1.038	1.049	1.047
Proxy-Hash-SL-0.3	1.050	1.045	1.034	1.051	1.066	1.064	1.054	1.070
Proxy-Hash-SL-0.4	1.098	1.042	1.058	1.049	1.073	1.063	1.068	1.069
Proxy-Hash-SL-0.5	1.105	1.070	1.061	1.049	1.081	1.082	1.069	1.071

Table B.5 Ratio between total road distance of the tree and real MST, the Great Lakes

Sample Size	616	1289	3063	6564	14056	26004
Euclidean	1.434	1.586	1.433	1.623	1.799	1.824
Proxy-Hash-SL-0.1	1.054	1.036	1.029	1.036	1.030	1.031
Proxy-Hash-SL-0.2	1.053	1.067	1.038	1.046	1.057	1.049
Proxy-Hash-SL-0.3	1.111	1.081	1.042	1.079	1.052	1.071
Proxy-Hash-SL-0.4	1.108	1.100	1.043	1.084	1.074	1.082
Proxy-Hash-SL-0.5	1.081	1.094	1.062	1.085	1.118	1.083

Figure B.3, B.4 and B.5 show the performance of our algorithm versus sample size and different ϵ values. Here Proxy-Hash-SL-ϵ refers to spanning tree constructed by Proxy-Hash-SL using ϵ as parameter.

Figure B.7 and B.8 show the number of distance computations that would be needed by our algorithm, versus sample size and A. As is the case with New York, it grows only slightly super-linearly with sample size.

Figure B.6, B.7 and B.8 show the number of distance evaluations done by the algorithm. The naive implementation this grows quadratically. We see that Proxy-Hash grows slightly superlinearly with input size.

(a) New York

(b) Bay Area

(c) The Great Lakes

(d) Seizure

Fig. B.2 Number of distance computation, versus sample size (log), Proxy-Hash-SL

Fig. B.3 Approximation ratio versus sample size and value for New York

Fig. B.4 Approximation ratio versus sample size and value for Bay Area

Fig. B.5 Approximation ratio versus sample size and value for the Great Lakes