On Scalable Algorithms and Algorithms with Predictions

Thomas Lavastida

May 11, 2022

Tepper School of Business
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Algorithms, Combinatorics, and Optimization.

To my parents, for their love and always believing in me even when I did not

Abstract

In recent years the massively increased availability of data has created significant interest in data-driven methods in engineering and business. This has created a need both for more scalable algorithms to process larger data sets as well as new methods for leveraging data in decision making and optimization. In this dissertation we consider both of these broader questions for a variety of relevant problems. The first part of this dissertation is concerned with the question of scalability via *parallel and distributed algorithms*, while the second part considers questions within the growing field of *algorithms with predictions*.

In Chapter 1, we consider the weighted longest common subsequence problem and its all-substrings generalization. This problem arises in computational biology applications, where there is a need to scale to larger inputs. We give efficient parallel algorithms which yield nearly optimal solutions for both problems.

Hierarchical clustering is a fundamental data analysis tool, and one of the most widely used clustering methods in practice. Typical approaches suffer from issues of scalability, either due to needing at least quadratic time or having an inherently sequential nature. To get around these barriers, we consider approximations. Our main results for Chapter 2 are efficient distributed algorithms which approximate the wide class of divisive k-clustering methods.

In *algorithms with predictions* we augment our usual algorithms with additional information representing (potentially erroneous) predictions about the input or desired solution. These predictions can be useful in resolving future uncertainty in the online setting or improving the average case running time of an algorithm when a problem must be solved repeatedly on similar inputs. In general, we want to design and analyze algorithms whose performance is tied to prediction error, as well as to show how to appropriately construct these predictions from past data.

Chapter 3 considers online load balancing with restricted assignments. We show the existence of predictions which can help guide the online algorithm in constructing a fractional assignment. These predictions are robust to small errors and they can be efficiently learned from past instances of the problem. To complete the result, we give an online rounding algorithm.

Finally, Chapter 4 considers the Hungarian algorithm for minimum cost bipartite matching. Usually this algorithm is given a naïve initial dual solution. We show, both theoretically and practically, that a *learned* initial dual solution can significantly improve the running time of this algorithm. This can be seen as using past instances of the problem to set a “warm-start” solution, which is more likely to be close to an optimal solution, and thus require fewer iterations to reach optimality.

Acknowledgments

First, I would like to thank my advisor, Ben Moseley, who originally took me on as a student when he was at Washington University in St. Louis. It has been a long journey to get here, and I could not have done it without Ben's endless support, optimism, and enthusiasm for research. Whether it was a paper rejection or preparing my job market talk, Ben has always been there to help me set my trajectory and offer feedback.

In addition to my advisor, I have been very luck to work with and learn from a wonderful group of co-authors throughout my years as a PhD student. I am thankful to have had the pleasure of working with Jeremy Buhler. Michael Dinitz, Sungjin Im, Silvio Lattanzi, Kefu Lu, R. Ravi, Sergei Vassilvitskii, Yuyan Wang, and Chenyang Xu.

Additionally, I would like to thank my committee, which included Ben Moseley, R. Ravi, Willem-Jan van Hoeve, and Cliff Stein. They provided me with useful feedback on my writing and talks, as well as gave me support and encouragement in pursuing an academic career.

I am grateful to my friends from LSU: Matthew, Daniel, Justin, Bo, and Bea. It has been a very interesting experience to see all of us move from being undergraduate students at LSU to each of us pursuing our respective paths professionally. I love that we have remained close even after moving all over the U.S. (or the world, in Daniel's case), and I hope that we can plan more meet-ups in the future.

My friends I have made while at Carnegie Mellon have made my time here unforgettable. I am grateful to Nam, Christian, Gerdus, Amin, Aleks, Michael, Ryo, Wenting, Yang, Arash, Goran, Neda, Ziye, Violet, Sagnik, Yuyan, Melda, Ozgun, Kyra, Su, Serim, Rudy, Mik, Anthony, Daniel, Sherry, and Yasamin for their friendship throughout my time as a PhD student. I am also grateful to my support group, which was very helpful to me while completing my PhD in the midst of a global pandemic.

Finally, I would like to thank my family. There have been several moments during my PhD where I have been anxious or stressed and my parents, Jorge and Maureen, have never failed to believe in me. I am immensely grateful for the love and support I have received from them, as well as from my sisters - Emily, Claire, Sara, and Gianna.

Contents

Abstract iv

Acknowledgments vi

Introduction 1

1 Parallel Approximation Algorithms for Weighted Longest Common Subsequence 7

1.1 Introduction 7
1.2 Preliminaries 11
1.3 All-Substrings Weighted Longest Common Subsequence 11
1.3.1 Divide-and-Conquer Strategy 11
1.3.2 Approximation via Sketching 12
1.3.3 Base Case Local Algorithm 13
1.4 Analysis of Approximation and Runtime 19
1.4.1 Quality of the Solution 19
1.4.2 Running Time 20

2 Distributed Algorithms for Hierarchical Clustering 23

2.1 Introduction 23
2.2 Preliminaries 25
2.3 A Framework for Parallelizing Hierarchical Clustering Algorithms 26
2.4 Algorithms and Theoretical Guarantees 27
2.4.1 Distributed Divisive k-Clustering 27
2.4.2 From Bounded Length Dependency Chains to Parallel Algorithms 29

3 Online Load Balancing with Predictions 31

3.1 Introduction 31
3.1.1 Related Work 34
3.2 Preliminaries 35
3.2.1 Problem Definition and Notation 35
3.2.2 ML Oracles 37
3.2.3 Predictions for Online Scheduling 38
3.3 A Robust Online Algorithm via Machine Weights 39

3.3.1 Constructing Fractional Solutions Online Using Learned Weights . . 39
3.4 Rounding Algorithm Overview 41
3.5 Online Rounding Algorithm & Analysis 43
3.5.1 Instance Transformation 44
3.5.2 Rounding A Single Class of Large Jobs 45
3.6 Lower Bounds for Online Rounding 54
3.6.1 Deterministic Lower Bound 55
3.6.2 Randomized Lower Bound 55
3.6.3 Learning the Weights 58
3.7 Existence of Weights for a Near Optimal Fractional Assignment 64
3.8 Proof of Rounding Theorem 67
3.9 Removing Knowledge of T 67

4 Speeding up the Hungarian Algorithm with Learned Duals 71
4.1 Introduction 71
4.1.1 Our Contributions 72
4.1.2 Related Work 74
4.1.3 Roadmap 75
4.2 Preliminaries 75
4.3 Faster Min-Weight Perfect Matching 76
4.3.1 Recovering a Feasible Dual Solution (Feasibility) 77
4.3.2 Seeding Hungarian with a Feasible Dual (Optimization) 79
4.3.3 Learning Optimal Advice (Learning) 82
4.4 Experiments 86
4.4.1 Running Time 90
4.5 Extending to b-Matching 90
4.5.1 Recovering a Feasible Dual Solution for b-Matching 93
4.5.2 Converting a Feasible Dual Solution to an Optimal Primal Solution 94
4.5.3 Learning the Dual Prices 96
4.6 Conclusion and Future Work 97

Conclusions 99

Bibliography 101

Introduction

In recent years the massively increased availability of data has created significant interest in data-driven methods in engineering and business. This has created a need both for more scalable algorithms to process larger data sets as well as new methods for leveraging data in decision making and optimization. In this dissertation we consider both of these broader questions for a variety of problems, including string comparison, hierarchical clustering, online scheduling, and bipartite matchings.

The first part of this dissertation is concerned with the question of scalability via *parallel and distributed algorithms*, while the second part considers questions within the growing field of *algorithms with predictions.*

Parallel and Distributed Algorithms

While data sets have greatly increased in size, the availability of large amounts of cheap computing resources has also increased through online cloud computing platforms such as Amazon Web Services [9], Microsoft Azure [112], and Google Cloud Platform [73]. Thus a common approach to achieving greater scalability is to utilize multiple computing resources in tandem. This could take the form of a multi-core CPU or a distributed network of several machines across which the computation takes place.

In order to design and analyze algorithms we need to consider models which capture aspects of these practical settings. In this dissertation we will describe algorithms within the (Parallel Random Access Machine (PRAM) [65] and Massively Parallel Computation (MPC) [90, 72, 36] models of computation. These models are designed to distill the essential components of the practical settings of a multi-core CPU or a distributed network of machines, respectively. This allows algorithm designers to devise new algorithms which can better leverage multiple computing resources to achieve greater scalability.

PRAM: In the PRAM model, there are a fixed number p of parallel processors with a shared memory where the input is initially stored. Issues of synchronization will not be important as our algorithms will parallelize naturally, so we will ignore them here. The main quantity of interest will be the parallel running time of our algorithms.

MPC: In the MPC model there is a collection of M machines each with their own local memory of size S. Initially, the input of size N is distributed across the M machines so that each machine has a different piece of the input. Computation is carried out in rounds. In a single round, each machine is allowed to carry out computation on data stored in its local memory, after which there is a synchronous communication step where machines are allowed to exchange information. The total amount communicated by each machine (both sending and receiving) should be $O(S)$. This model is widely used to capture the class of algorithms that scale in programming frameworks such as Spark and MapReduce.

We will require that $S \cdot M = \tilde{O}(N)$, which enforces that we cannot communicate all of the data to one machine and apply a sequential algorithm. Here the main quantity of interest is the number of rounds needed to carry out a computation under the previous constraints, which we would like to be polylogarithmic in N. A typical setting of the parameters has $S = \tilde{O}(N^\delta)$, for some constant $\delta \in (0, 1)$. This way we have both S and M sublinear in N.

Breaking Dependencies: Ideally, when designing a parallel algorithm, we want to break up a computation into several independent parts which can be handled in parallel by separate machines/processors. One of the main challenges in designing an efficient parallel algorithm is dealing with *chains of dependencies*. Dependencies occur when one step in a computation depends on a previous one. If an algorithm has a long chain of dependencies, then this creates a bottleneck for parallelizing it. For the problems considered in this dissertation, standard algorithms will suffer from this problem.

To understand this issue more concretely, let's consider the longest common subsequence (LCS) problem and its standard dynamic programming algorithm. In this problem, there are strings x and y over a common finite alphabet Σ. We will let the length of x be n and the length of y be m. The goal is to find a subsequence that is common to both strings that is as long as possible. A standard dynamic programming algorithm computes an optimal solution in time $O(nm)$ by setting up the following recurrence. Let $L(i, j)$ be the length of the longest common subsequence between $x[1 : i]$ and $y[1 : j]$ (where $x[1 : i]$ refers to the substring of x consisting of the first i characters). Ignoring base cases, for $i, j > 1$ we recursively compute:

$$L(i, j) = \max\{L(i-1, j), L(i, j-1), L(i-1, j-1) + \mathbf{1}_{\{x[i]=y[j]\}}\}.$$

Since the goal is to finally compute $L(n, m)$, observe from the recurrence above that doing so will have dependency chains of length $\Theta(n + m)$, thus making it difficult to parallelize this algorithm.

Thus, a key challenge for algorithm designers is to find algorithms with significantly smaller chains of dependencies which can be implemented efficiently in parallel and distributed settings. In this dissertation, we will do this through the use of approximation. That is, rather than insisting on producing an optimal solution or making exact decisions, we will relax our algorithms and allow them to produce nearly optimal solutions or make approximate decisions throughout their execution. A natural question to ask is: when can approximations be leveraged to break long chains of dependencies and allow us to achieve better scalability? We will answer this question affirmatively for a weighted version of the LCS problem described above and also for divisive hierarchical clustering methods.

Chapter 1: Parallel Approximation Algorithms for Weighted Longest Common Subsequence

In the weighted longest common subsequence (WLCS) problem we are given strings x and y over a common finite alphabet Σ (of length n and m, respectively) as well as a non-negative scoring function $f : \Sigma \times \Sigma \to \mathbb{N}$. The objective is to output a correspondence with maximum total weight. A correspondence between x and y is a sequence of index pairs $(i_1, j_1), (i_2, j_2), \ldots, (i_\ell, j_\ell)$ such that $i_k < i_{k+1}$ and $j_k \leq j_{k+1}$ for all k. The weight of a correspondence is given by $\sum_{k=1}^{\ell} f(x[i_k], y[j_k])$. This problem generalizes the longest common subsequence problem and a simple modification of the dynamic programming algorithm described above can be used to find an optimal solution in $O(nm)$ time (sequentially).

This problem has applications in bioinformatics, where the strings x and y correspond to DNA sequences which may be millions to billions of characters long. Thus there is a need to develop more scalable approaches since the sequential $O(nm)$ running time may be prohibitive at these scales.

In Chapter 1, we develop efficient parallel approximation algorithms for this problem. Our algorithms take in a parameter $\epsilon \in (0, 1)$ and will output a solution with weight at least $1 - \epsilon$ times the weight of an optimal correspondence. The main result of this chapter is a parallel algorithm with running time

$$O\left(\frac{\sigma m n}{p} + \frac{m}{\epsilon^2} \log^2(\sigma m) \log^2(n) \log(p)\right),$$

where σ is an upper bound on the scoring function (which is usually small in practice). We also note that our algorithms can be adapted to run in $O(\log n)$ rounds in the MPC setting when $m = O(\sqrt{n})$.

Chapter 2: Distributed Algorithms for Hierarchical Clustering

In hierarchical clustering, we are given a set of n points S along with a dissimilarity function $d : S \times S \to \mathbb{R}$ and the goal is to output a binary tree on n leaves, where each leaf corresponds to one of the points in S. Internal nodes in the tree represent clusters which contain all of the leaf nodes in the sub-tree rooted at the node. Ideally, more dissimilar points are separated near the top of the tree, while more similar points are separated closer to the bottom of the tree.

A natural method for constructing a hierarchical clustering tree is to first use a clustering method such as k-means with $k = 2$ to divide the set S into two parts, and then continue recursively on each part. While it is known how to compute a k-means clustering efficiently in the MPC setting, adapting this divisive algorithm to the MPC setting is not immediate. It is possible for the k-means clustering to produce unbalanced clusters in each step, causing long chains of sequential dependencies and resulting in an inefficient parallel algorithm. In Chapter 2, we will show that we can overcome this issue while retaining an $O(1)$-approximate solution to the clustering problem in each step. In particular, we will give divisive hierarchical clustering algorithms in the MPC setting running in $O(\log n)$ rounds under mild assumptions, which guarantee that the split at each step is an $O(1)$-approximation for the clustering cost used by the divisive algorithm.

Algorithms with Predictions

Recent advances in machine learning have revolutionized how problems in computer vision, natural language processing, and perception are addressed. Improvements to predictive models and increased availability of large data-sets have contributed to this rapid progress. One might also ask if it's possible to leverage machine learning and predictive models to improve the design of algorithms for optimization problems. Machine learning models use past data to infer structure in the (unknown) target distribution that allow them to make accurate predictions on new examples from the same distribution. Could we use past instances of an optimization problem in order to learn something which can assist in solving new instances of the problem, ideally with some sort of improved performance?

Following a rekindled interest in beyond worst case analysis of algorithms [129], there has been significant effort in developing methods for incorporating machine learning and data-driven methods into algorithm design for combinatorial problems. This area has come to be known as *learning-augmented algorithms* or *algorithms with predictions* [116]. In this dissertation we will use the latter term to refer to this area. The aim of this area is to design algorithms which incorporate predictions so that the algorithm can go beyond worst-case lower bounds when the prediction is accurate and retain similar worst-case performance otherwise.

A natural setting where predictions and learning can help is in *online optimization*. In online optimization, each piece of the problem is revealed one at a time and the algorithm must commit to a decision before seeing more of the problem. For example, in online scheduling the algorithm must decide how to schedule a newly arrived job before seeing future jobs. This commitment to a decision before seeing the entire input means that we usually expect the online algorithm to find sub-optimal solutions. Thus we want to quantify and upper bound how sub-optimal the algorithm is. Usually this is done in a worst case sense. For some instance $\mathcal{I}$ of a problem, say that $\text{ALG}(\mathcal{I})$ is the cost incurred by an online algorithm and $\text{OPT}(\mathcal{I})$ is the cost of an optimal solution in hindsight (which knows the entire input sequence). Then we say that an algorithm is c-competitive (or has competitive ratio c) if for all instances $\mathcal{I}$ we have $\text{ALG}(\mathcal{I}) \leq c \cdot \mathcal{I}$. Since this is worst-case, usually one can show strong lower bounds on possible values of c.

In the setting of online algorithms with predictions, we hope that accurate predictions can allow an online algorithm to go beyond worst-case lower bounds on the competitive ratio, but perform nearly the same as in the worst-case setting when the predictions are very inaccurate, with a smooth degradation in-between. This line of work was initiated by Lykouris and Vassilvitskii [107, 108], who studied the online caching problem in the presence of predictions. On the arrival of each page request, they assume that the online algorithm is also given a prediction of when it will arrive again in the future. They quantify the competitiveness of their algorithm in terms of the ℓ_1-distance between the vector of predicted arrival times and the vector of true arrival times. More recently there has been an explosion of work in this area, covering many classic online settings such as the ski rental problem [126, 69, 11], further work on caching [128, 85, 148, 32], page migration [82], scheduling [126, 115, 20], online covering [31], bipartite matching [15, 103, 1], metrical task systems [14], and Steiner tree [22, 150]. In terms of online algorithms with predictions, this dissertation will consider online load balancing with restricted assignments, a classic online scheduling problem with strong lower bounds in the worst-case setting.

In addition to competitive ratios in the online setting, there has been significant interest in using predictions to improve other aspects of an algorithm's performance, such as running time and space complexity. For example, Kraska et al. [94] use learning to improve query times in index structures, a standard tool for data retrieval. Mitzenmacher [114] uses predictions to improve the space complexity of bloom filters. Hsu et al. [80] using predictions to improve the errors in sketches for heavy-hitters in the streaming setting. Chmiela et al. [48] consider learned primal-heuristic schedules to improve primal performance in mixed integer programming solvers, which can also help to improve running time. In this dissertation we will study how to improve the running time for weighted bipartite matching algorithms through predictions of the optimal dual solution.

Another related area is data-driven algorithm design [24, 75, 28, 27, 26, 25]. Here a parameterized family of algorithms is considered, and the goal is to use data on past problem instances to learn a setting of the parameters for the algorithm so that it performs well on instances drawn from the same population as the past problem instances used in learning. Usually, the main goal is to understand the *sample complexity* of learning the parameter, i.e. how much data is needed. We will make use of ideas from this area in order to show that the predictions we propose are also *learnable*. Additionally, work within the algorithms with predictions community has also recognized that learnability is an important and interesting property for a prediction to have [11, 10, 55].

Research Questions: In this dissertation we will consider the setting of algorithms with predictions in the context of optimization problems. This raises several questions to address when faced with a concrete optimization problem:

1. What quantity should be predicted? This should depend on the structure of the problem at hand.

2. How do we incorporate the predictions in the design of the algorithm? Ideally, more accurate predictions should yield improved performance.

3. How do we construct the prediction from past problem instances? Can we guarantee the prediction we learn will generalize to new problem instances?

In the final two chapters of this dissertation, we will consider all of these questions for two classic optimization problems: online makespan minimization and improving running time for minimum cost bipartite matchings.

Chapter 3: Online Load Balancing with Predictions

In this chapter, we will consider online makespan minimization with restricted assignments in the algorithms with predictions setting. In this problem, there is a collection of n jobs which arrive online to be scheduled on m machines. Upon a job j's arrival, it reveals its size p_j and a subset $N(j) \subseteq [m]$ of feasible machines where it can be assigned. The algorithm must irrevocably assign the job to some machine $i \in N(j)$ before seeing future jobs. The objective is to minimize the maximum load of a machine, where the load of machine i is the total size of jobs assigned to it.

Online makespan minimization with restricted assignments is a fundamental problem in online scheduling and well understood from the perspective of worst-case analysis. Notably, all online algorithms are $\Omega(\log m)$-competitive and the natural greedy algorithm which assigns each job to the least loaded feasible machine is $O(\log m)$-competitive [21]. We will be interested in finding quantities to predict that succinctly captures the structure of an instance, so that highly accurate predictions can go beyond this worst-case lower bound. Our approach will construct *fractional* assignments, and thus we also study the problem of rounding a fractional assignment online in order to be competitive with the in-hindsight makespan of the fractional assignment, providing both upper and lower bounds for this online rounding problem. Additionally, we will show that our proposed prediction is in some sense learnable from past data.

Chapter 4: Speeding up the Hungarian Algorithm with Learned Duals

Another natural setting where predictions can help is in the context of improving running times. In addition to ensuring that our algorithm has a good worst-case running time, we can use learned predictions to tailor our algorithm to the inputs that are typically seen in practice. In Chapter 4, we do this for the minimum cost perfect matching problem in bipartite graphs. In this problem, there is a bipartite graph $G = (V, E)$ with integer costs c on the edges of the graph. The goal is to output a perfect matching M which minimizes the cost $c(M) = \sum_{e \in M} c_e$.

A standard algorithm for solving this problem is the Hungarian algorithm [97], which is based on updating dual variables for a linear programming formulation of the problem in order to reach optimality. Usually, these dual variables are initialized in a naïve way in order to guarantee dual feasibility. The main idea for this chapter is to predict initial values for the dual variables with the hope that they are likely to be closer to the optimal dual solution for a new instance, and thus require fewer iterations of the Hungarian method to reach optimality. There are several challenges we have to address:

1. What if the predicted duals are infeasible for the new problem instance?

2. Does a more accurate prediction of the duals lead to an improved running time?

3. How do we use past data to predict values for the dual variables?

We will answer all three of these questions, which will give us an end-to-end framework for incorporating predicted duals into the Hungarian algorithm.

In addition to these theoretical results, we conduct experiments based on both synthetic and real data which demonstrate the capability of learned duals to significantly outperform standard initialization methods for the Hungarian algorithm in terms of running time.

Chapter 1

Parallel Approximation Algorithms for Weighted Longest Common Subsequence

This chapter is based on ”A Scalable Approximation Algorithm for Weighted Longest Common Subsequence” [42], which appeared in the proceedings of the 27th International European Conference on Parallel and Distributed Computing (EURO-PAR 2021). Collaborators on the project were Jeremy Buhler, Kefu Lu, and Benjamin Moseley.

1.1 Introduction

Technologies for sequencing DNA have improved dramatically in cost and speed over the past two decades [125], resulting in an explosion of sequence data that presents new opportunities for analysis. To exploit these new data sets, we must devise scalable algorithms for analyzing them. A fundamental task in analyzing DNA is comparing two sequences to determine their similarity.

A basic similarity measure is weighted longest common subsequence (WLCS). Given two strings x and y over a finite alphabet Σ (e.g. $\{\texttt{A,C,G,T}\}$), a *correspondence* between them is a set of index pairs $(i_1, j_1) \ldots (i_\ell, j_\ell)$ in x and y such that for all $k < \ell$, $i_k < i_{k+1}$ and $j_k < j_{k+1}$. A correspondence need not use all symbols of either string. We are given a non-negative scoring function $f : \Sigma \times \Sigma \to \mathbb{N}$ on pairs of symbols, and the goal is to find a correspondence (the WLCS) that maximizes the total weight $\sum_{k=1}^{\ell} f(x[i_k], y[j_k])$. We assume, consistent with actual bioinformatics practice [138], that the maximum weight σ returned by f for any pair of symbols is a small constant [88], so that the maximum possible weight for a correspondence between sequences is proportional to their length. WLCS is a special case of the weighted edit distance problem [120] in which match and mismatch costs are non-negative and insertion/deletion costs are zero. This problem is sufficient to model similarity scoring with a match bonus and mismatch and gap penalties, provided we can subsequently normalize alignment weights by the lengths of the two sequences [141]. If f scores +1 for matching symbol pairs and 0 for all others, the problem reduces to unweighted LCS.

A generalization of WLCS is the *all-substrings WLCS* or *AWLCS* problem. In this variant, the goal is to compute a matrix H such that $H[i,j]$ is the weight of a WLCS between the entire string x and substring $y[i..j]$. This "spectrum" of weights can be used to infer structure in strings, such as approximate tandem repeats and circular alignments [135]. Of course, H includes the weight of a WLCS between the full strings x and y as an entry.

Throughout this chapter, we let $|x| = n$, $|y| = m$ and assume that $n \geq m$.

Sequential Methods The WLCS problem, like the unweighted version, can be solved by dynamic programming in time $O(nm)$. In particular, the well-known Needleman-Wunsch algorithm [120] for weighted edit distance, which is the basis for many practical biosequence comparison tools [137, 132, 133, 134], solves the WLCS problem as a special case. Sub-quadratic time algorithms are also known for the WLCS problem based on the "Four Russians" technique [110], which works for integer weights. In addition, there is the work of Crochemore et al that works for unrestricted weights and also achieves sub-quadratic time [51]. At the same time, the sequential complexity of the LCS and WLCS problem is well understood - results in fine-grained complexity give strong lower bounds assuming the Strong Exponential Time Hypothesis, see e.g. [40, 39] and [2].

Schmidt [135] showed that AWLCS, which naively requires much more computation than WLCS, can be solved in time $O(nm \log m)$ as a special case of all-substrings weighted edit distance. Alves et al. reduced this cost to $O(nm)$ for the special case of unweighted all-substrings LCS (ALCS) [8].

Parallel Methods One way to solve large WLCS problems more efficiently is to parallelize their solution. Krusche and Tiskin [96] study parallelization of standard dynamic programming algorithms for LCS. However, the straightforward dynamic programming approaches for LCS and WLCS do not easily parallelize because they contain irreducible chains of dependent computations of length $\Theta(n + m)$. The fastest known parallel algorithms for these problems instead take a divide-and-conquer approach (such as [16]), combining the all-substrings generalization of LCS with methods based on max-plus matrix multiplication as we will describe.

Let x_1 and x_2 be two strings, and let H_1 and H_2 be AWLCS matrices on string pairs (x_1, y) and (x_2, y), respectively. Defining matrix multiplication over the ring $(\max, +)$, $H_1 \times H_2$ is the AWLCS matrix for strings $x_1 \cdot x_2$ and y [142]. Hence, we can compute the AWLCS matrix for the pair (x, y) on p processors by subdividing x into p pieces x_k, recursively computing matrices H_k for each x_k with y, and finally multiplying the H_k together. Given a base-case algorithm to compute AWLCS in time $B(m, n)$ and an algorithm to multiply two $m \times m$ AWLCS matrices in time $A(m)$, this approach will run in time $B(m, \frac{n}{p}) + A(m) \log p$.

Fast multiplication algorithms exist that exploit the *Monge property* of AWLCS matrices: for all $1 \leq i < k \leq m$ and $1 \leq j < \ell \leq n$, $H[i,j] + H[k,\ell] \leq H[i,\ell] + H[k,j]$. Tiskin [142] showed that for the special case of unweighted ALCS, $A(m) = O(m \log m)$, yielding an overall time of $O\left(\frac{mn}{p} + m \log m \log p\right)$.

For AWLCS, $A(m) = O(m^2)$ using an iterated version of the SMAWK algorithm [4, 131].

No faster multiplication algorithm is known for the general case. Practically subquadratic multiplication has been demonstrated for specific scoring functions f [131], but the performance of these approaches depend on f in a difficult-to-quantify manner. In [127] a complex divide-and-conquer strategy was used to achieve an optimal running time for the pairwise sequence alignment problem, which is similar but more general than our problem. In our work we use an alternative divide-and-conquer strategy to obtain a fast parallel algorithm.

Contributions: This chapter introduces a new approach to parallelizing AWLCS and therefore WLCS. We introduce algorithms that are $(1-\epsilon)$ approximate. Our algorithm's running time scales with $o(m^2)$ in the PRAM setting as the number of processors increases.

The new algorithm is our main contribution. Our algorithm sketches a sequential dynamic program and uses a divide-and-conquer strategy which can be parallelized. This sketch comes with a cost of approximating the objective to within a $1-\epsilon$ factor for any parameter $\epsilon \in (0,1)$. By relaxing the algorithm's optimality guarantee, we are able to obtain subquadratic-time subproblem composition by building on recent results on parallelizing dynamic programs for other problems [81]. Additionally, we develop and utilize a new base case algorithm for AWLCS that takes advantage of the small range of weights typically used [88]. The following theorem summarizes our main result.

Theorem 1.1.1. *Let W be the largest possible correspondence weight and let p be the number of processors. For any $\epsilon \in (0,1)$, there is a PRAM algorithm running in time $O(B(m, \frac{n}{p}) + \frac{m}{\epsilon^2}\log^2(W)\log^2(n)\log(p))$ that computes a $(1-\epsilon)$-approximate solution to the WLCS problem.*

Using Schmidt's algorithm ($B(m,n) = O(mn\log m)$) for the base case, we obtain a parallel algorithm with running time:

$$O\left(\frac{mn\log m}{p} + \frac{m}{\epsilon^2}\log^2 W \log^2 n \log p\right),$$

where W is the largest possible correspondence weight between strings(which, by our assumption of bounded weights, is $O(\min(n,m))$). As mentioned, this is the first parallel algorithm for weighted LCS for which the running time scales as $o(m^2)$, and also the fastest $(1-\epsilon)$-approximation algorithm for weighted LCS in BSP. In contrast, previous methods' running times have a $\Theta(m^2)$ term that does not diminish as the number of processors p increases.

Using Schmidt's algorithm for the base case dominates the running time. We would like to improve the $O(mn\log m)$ running time of the base case to get as close to $O(mn)$ as possible. We develop an interesting alternative base case algorithm by extending Alves's $O(mn)$ algorithm for ALCS [8] to the weighted case of AWLCS. This is our second major contribution. This algorithm, like our overall divide-and-conquer strategy, exploits the small range of weights typically used by scoring functions for DNA comparison.

Theorem 1.1.2. *Let σ be the highest weight produced by the scoring function f. There is a sequential algorithm running in time $O(\sigma nm)$ time for computing an implicit representation of the AWLCS matrix using space $O(\sigma m)$.*

Using this algorithm as the base case in Theorem 1.1.1, we achieve an overall running time of $O(\frac{\sigma mn}{p} + \frac{m}{\epsilon^2}\log^2(\sigma m)\log^2(n)\log(p))$.

Algorithmic Techniques We leverage two main techniques. The first is *parallelizing a natural dynamic program* for a problem via sketching. Let $C(i,j)$ be the weight of a WLCS between $x[1:n]$ and $y[i:j]$; we want to compute this quantity for all $1 \leq i < j \leq m$. One may add a third index to specify $C_k(i,j)$, the weight of a WLCS between $x[1:k]$ and $y[i:j]$. C_k can be computed via the following recurrence: $C_k(i,j) = \max\{C_{k-1}(i,j), C_k(i,j-1), C_{k-1}(i,j-1) + f(x[k],y[j])\}$. But this recurrence is both inefficient, requiring time $O(nm^2)$, and difficult to parallelize, with dependent computation chains of size $\Omega(n+m)$.

To improve efficiency, we abandon direct computation of $C(i,j)$ and instead compute some $D(i,w)$ which is subsequently be used to derive the entries of $C(i,j)$. $D(i,w)$ is the least index j s.t. there exists a correspondence of weight at least w between $y[i:j]$ and $x[1:n]$. We compute and store $D(i,w)$ only for values w that are powers of $1+\epsilon'$ for some fixed $\epsilon' > 0$. This sketched version of D effectively represents the $O(m^2)$ sized matrix C using $O(m\log_{1+\epsilon'} m\sigma)$ entries. Although our sketching strategy is not guaranteed to find the optimal values $C(i,j)$, we show that it exhibits bounded error as a function of ϵ'.

A straightforward computation of $D(i,w)$ entails long chains of serial dependencies. Thus, we use a divide-and-conquer approach instead. Let $D_{r_1,r_2}(i,w)$ store the the minimum index j s.t. a correspondence of weight at least w exists between $y[i:j]$ and $x[r_1:r_2]$. We will show how to compute $D_{r_1,r_3}(i,w)$ given $D_{r_1,r_2}(i,w')$ and $D_{r_2+1,r_3}(i,w')$ for values $w' \leq w$. If we compute D matrices for non-overlapping substrings of x in parallel and double the range of x covered by each D matrix at each step, we can compute $D_{1,n}(i,w)$ in a logarithmic number of steps.

In realistic applications, we seek to compare sequences with millions of DNA bases; the number of available processors is small in comparison, that is, $p \ll \min(n,m)$. Speedup is therefore limited by the base-case work on each processor, which must sequentially solve an AWLCS problem of size roughly $m \times \frac{n}{p}$. Solving these problems using Schmidt's algorithm, which is insensitive to the magnitude of weights, takes time $O(\frac{n}{p}m\log m)$. However, Schmidt's algorithm involves building complex binary trees which proved to have high overhead in practice. Our second main technique is developing a weight-sensitive AWLCS algorithm utilizing an efficient and compact implicit representation.

We show that if the scoring function f assigns weights at most σ to symbol pairs, the matrix $C(i,j)$ can be represented implicitly using only $O(m\sigma)$ storage rather than $O(m^2)$. Moreover, we can compute this representation in sequential time $O(\frac{n}{p}m\sigma)$. The algorithm computes and stores values of the form $h_s(j)$, which is the least index i such that $C(i,j) \geq C(i,j-1)+s$, for $1 \leq s \leq \sigma$. These h values indicate where there is an increase of s in the optimal correspondence weight when the index j increases. The key to the technique is showing that these values contain information for reconstructing C and how to compute them efficiently without complex auxiliary data structures.

Roadmap Section 1.3.1 presents the main dynamic program which can be parallelized via a divide-and-conquer strategy, while Section 1.3.2 shows how to use sketching to make this step time and space efficient while retaining $(1-\epsilon)$-approximate solutions.

Section 1.3.3 presents our new algorithm for AWLCS which we use as an efficient local base case algorithm on each processor. Finally, Section 1.4 completes our analysis.

1.2 Preliminaries

We denote by $x[i : j]$ the contiguous substring of x that starts at index i and ends at index j. The goal of AWLCS is to find correspondences of maximum weight between $x[1 : n]$ and $y[i : j]$ for all $1 \leq i \leq j \leq m$. We develop a method to obtain the *weights* of the desired correspondences; the alignments can be recovered later by augmenting the recurrence to permit traceback of an optimal solution. However, for AWLCS, the weights alone suffice for many applications [135]. Finally, we denote by W the highest possible weight of a WLCS between x and y, which we assume to be $O(\sigma \min(n, m))$. Here $\sigma = \max_{c,c' \in \Sigma} f(c, c')$ is the maximum possible weight of matching two characters. We note that in practice, σ is a constant and typically less than 20.

We now define two key matrices utilized in the design of our algorithms. $C(i, j)$ will denote the maximum weight of a correspondence between x and $y[i : j]$. The AWLCS problem seeks to compute $C(i, j)$ for all $1 \leq i < j \leq m$. An alternative way to view these weights is via the matrix D, where we swap the entry stored in the matrix with one of the indices. Let $D(i, w) = \min\{j \mid C(i, j) \geq w\}$. If no such j exists, we define $D(i, w) = \infty$. D stores essentially the same information as C; a single entry of $C(i, j)$ can be queried via the matrix D in time $O(\log W)$ by performing a binary search over possible values of w. However, the matrix D will be a substantially more compact representation than C once we introduce our sketching strategy.

1.3 All-Substrings Weighted Longest Common Subsequence

Here we present our algorithm for AWLCS. Following the divide-and-conquer strategy of prior work, we initially divide the string x equally among the processors, each of which performs some local computation using a base-case algorithm to solve AWLCS between y and its portion of x, yielding a solution in the form of the D matrix defined above. We then combine pairs of subproblem solutions iteratively to arrive at a global solution. We first describe the algorithm's divide and combine steps while treating the base case as a black box, then discuss the base-case algorithm.

1.3.1 Divide-and-Conquer Strategy

Let D_{r_1,r_2} be the D matrix resulting from the AWLCS computation between strings $x[r_1 : r_2]$ and y. Our goal is to compute $D_{1,n}$, which encompasses all of x and y.

Our algorithm first divides x into p substrings of length $\frac{n}{p}$, each of which is given to one processor along with the entire string y. We assume that consecutive substrings of x are given to consecutive processors in some global linear processor ordering. If a processor is

Algorithm 1 Combining Subproblems

```
procedure COMBINE($D_{r_1,r_2}, D_{r_2+1,r_3}$)
    for $i = 1$ to $m$ do
        for $w = 0$ to $W$ do
            $D_{r_1,r_3}(i, w) \leftarrow \infty$
            for $w_1 = 0$ to $w$ do
                $w_2 \leftarrow w - w_1$
                $j' \leftarrow D_{r_1,r_2}(i, w_1)$
                $j \leftarrow D_{r_2+1,r_3}(j' + 1, w_2)$
                $D_{r_1,r_3}(i, w) = \min(D_{r_1,r_3}(i, w), j)$
            end for
        end for
    end for
end procedure
```

given a substring $x[r_1 : r_2]$, it computes the subproblem solution D_{r_1,r_2} using our new local, sequential base-case algorithm described in Section 1.3.3. It then remains to combine the p subproblem solutions to recover the desired solution $D_{1,n}$. We compute $D_{1,n}$ in $O(\log p)$ rounds. In the j'th round, the algorithm computes $O(2^{\log(p)/j})$ subproblem solutions, where each solution combines two sub-solutions from adjacent sets of 2^{j-1} consecutive processors.

Let D_{r_1,r_2} and D_{r_2+1,r_3} be adjacent sub-solutions obtained from previous iterations. We combine these solutions to obtain D_{r_1,r_3}. To compute $D_{r_1,r_3}(i, w)$, we consider all possible pairs w_1, w_2 for which $w = w_1 + w_2$. For each possible w_1, we use the solution of the first subproblem to find the least index j' for which there exists a correspondence of weight w_1 between $x[r_1 : r_2]$ and $y[i : j']$. We then use the solution of the second subproblem to find the least j such that a correspondence of weight $w_2 = w - w_1$ exists between $x[r_2 + 1 : r_3]$ and $y[j' + 1 : j]$. (Clearly, $j \geq j'$.) These two correspondences use non-overlapping substrings of x and y and can be combined feasibly. The exact procedure can be found in Algorithm 1.

1.3.2 Approximation via Sketching

Algorithm 1 solves the AWLCS problem exactly; the cost to combine two subproblems is $O(mW^2)$. For unweighted ALCS, $W = m$; the combine step is $O(m^3)$. To overcome this cost, we *sketch* the values of w. Sketching reduces the number of distinct weights considered from W to $O(\log W)$ and hence reduces the cost to combine two subproblems from $O(mW^2)$ to $O(m \log^2 W)$. We analyze its precise impact on solution quality and overall running time in Section 1.4.

Our sketching strategy fixes a constant $\epsilon > 0$ and sets $\beta = 1 + \frac{\epsilon}{\log n}$. Define $D^*(i, s)$ to be the least j such that there exists a correspondence between x and $y[i : j]$ with weight $w \geq \lfloor \beta^s \rfloor$. Define $D^*_{r_1,r_2}$ analogously to D_{r_1,r_2} for substrings of x. To compute $D^*_{r_1,r_3}$

from $D^*_{r_1,r_2}$ and $D^*_{r_2+1,r_3}$, we modify the algorithm described above as follows. For each power s s.t. $\lfloor \beta^s \rfloor \leq W$, we consider each power $s_1 \leq s$ and compute the least s_2 such that $\lfloor \beta^{s_1} \rfloor + \lfloor \beta^{s_2} \rfloor \geq \lfloor \beta^s \rfloor$. Let $j' = D^*_{r_1,r_2}(i, s_1)$ and $j = D^*_{r_2+1,r_3}(j'+1, s_2)$. Then there exist non-overlapping correspondences with weights at least β^{s_1} and β^{s_2}, and hence a combined correspondence of weight at least β^s, between $x[r_1 : r_3]$ and $y[i : j]$. We take $D^*_{r_1,r_3}(i, s)$ to be the least j' that results from this procedure. In Section 1.4, we formally show that this sketching strategy preserves $(1 - \epsilon)$-approximate solutions and analyze the runtime and space usage of our algorithm.

1.3.3 Base Case Local Algorithm

We now describe a sequential algorithm, inspired by the work of [7], to obtain the initial matrices $D_{r_1,r_2}(i, w)$ for each individual processor.

In theory, one could continue the divide and conquer approach on each local machine until the entry to compute is of the form $D_{r_1,r_1+1}(i, w)$, yielding a simple base case to solve. However, this procedure proves computationally inefficient with a fixed number p of processors. Instead, we propose a different base case algorithm for computing $D_{r_1,r_2}(i, w)$ which better fits our setting.

For this section, we will drop the indices r_1 and r_2 and create an algorithm for computing D for strings x and y. Each processor applies this same algorithm, but to different substrings $x[r_1 : r_2]$.

The algorithm works in two steps. First, we calculate two sequences of indices, referred to as the h- and v-indices. Then, we use these indices to compute $D(i, w)$ for all desired i, w. Intuitively, these indices give compact information about the structure of the C matrix (and hence the D matrix), specifically the magnitude of change between weights in adjacent rows and columns of C.

Definition of the Indices Recall the definition of the AWLCS matrix C, and let C^ℓ be the C matrix corresponding to the strings $x[1 : \ell]$ and y. Before proceeding with the definition of the h- and v-indices, we note a lemma concerning the *Monge* properties of C^ℓ. These properties are well-known; see, e.g., [7].

Lemma 1.3.1. *For any triple of indices i, j, ℓ, $C^\ell(i-1, j-1) + C^\ell(i, j) \geq C^\ell(i-1, j) + C^\ell(i, j-1)$, and $C^\ell(i-1, j) + C^{\ell-1}(i, j) \geq C^{\ell-1}(i-1, j) + C^\ell(i, j)$.*

The following corollaries result from rearranging terms in the previous lemma.

Corollary 1.3.2. *For any i, j, ℓ, $C^\ell(i, j) - C^\ell(i, j-1) \geq C^\ell(i-1, j) - C^\ell(i-1, j-1)$.*

Corollary 1.3.3. *For any i, j, ℓ, $C^\ell(i, j) - C^{\ell-1}(i, j) \leq C^\ell(i-1, j) - C^{\ell-1}(i-1, j)$.*

We now consider the implications of Corollary 1.3.2. Fix i, j and ℓ with $C^\ell(i, j) - C^\ell(i, j-1) = s$ for some s. This s is the difference in WLCS weight if the second string is allowed one extra character at its end ($y[j]$), since it is comparing $x[1 : \ell]$ with either $y[i : j]$ or $y[i : j-1]$. The corollary states that this difference is only greater for a substring of y that starts at $i' > i$ instead of i. Therefore, for each pair of fixed j, ℓ, there exists some minimal i such that $C^\ell(i,j) - C^\ell(i,j-1)$ is *first* greater than s, as it will be true for all $i' > i$. For different values of s, there are possibly different corresponding i which are minimal. Similar implications can be derived from Corollary 1.3.3.

Using this insight, we can define the h-indices and v-indices. These values $h_1, \dots, h_\sigma$ and $v_1, \dots, v_\sigma$ are the key to our improved base case algorithm. For $s \in [\sigma]$, $h_s(\ell, j)$ is the smallest index i such that $C^\ell(i,j) \geq C^\ell(i,j-1) + s$. That is, each $h_s(\ell,j)$ for a fixed ℓ and j marks the row of C^ℓ where we start to get a horizontal increment of s between columns $j-1$ and j. The v-indices are slightly different; $v_s(\ell,j)$ is the smallest index i such that $C^\ell(i,j) < C^{\ell-1}(i,j) + s$. The v-indices mark the row where we stop getting a vertical increment of s in column j between $C^{\ell-1}$ and C^ℓ. The entire matrix C^ℓ can be computed recursively as a function of the indices as follows:

$$C^\ell(i,j) = \begin{cases} C^\ell(i,j-1) & i < h_1(\ell,j) \\ C^\ell(i,j-1) + s & h_s(\ell,j) \leq i < h_{s+1}(l,j) \\ C^\ell(i,j-1) + \sigma & h_\sigma(\ell,j) \leq i \end{cases} \tag{1.1}$$

$$C^\ell(i,j) = \begin{cases} C^{\ell-1}(i,j) + \sigma & i < v_\sigma(\ell,j) \\ C^{\ell-1}(i,j) + s & v_{s+1}(\ell,j) \leq i < v_s(l,j) \\ C^{\ell-1}(i,j) & v_1(\ell,j) \leq i \end{cases} \tag{1.2}$$

The h and v-indices provide an efficient way to compute the entries in $D(i,w)$. If we can compute $C^\ell(i,j)$ for all i,j, then $D(i,w)$ is the smallest j for which $C^\ell(i,j) \geq w$. The indices actually correspond to a recursive definition of the values of $C^\ell(i,j)$.

The following intuition may help to interpret the h-indices. consider $h_1(\ell,j)$ for a fixed ℓ and j. This is the smallest value of i for which $C^\ell(i,j)$ exceeds $C^\ell(i,j-1)$ by at least 1. Suppose we compare the best WLCS of x and $y[i:j-1]$ against that of x and $y[i:j]$. There is a gain of one character (the last one) in the second pair of strings, so the second WLCS might have more weight. There is a unique value $h_1(\ell,j)$ of i for which the difference in weight first becomes ≥ 1. The uniqueness of this value can be inferred from Corollary 1.3.2.

The increment in the WLCS weight due to adding $y[j]$ may become greater as i increases, i.e., as we allow fewer opportunities to match x to earlier characters in y. However, the increment cannot exceed σ, the greatest possible weight under f of a match to $y[j]$. Note that if $h_s(\ell,j) \geq j$, there is no index fulfilling the condition since the substring $y[h_s(\ell,j):j]$ has no characters.

Given the h-indices for every ℓ, j, we may compute $C^\ell(i,j)$ for any fixed ℓ as follows. $C^\ell(i,i) = 0$ by definition, and $C^\ell(i,j+1)$ can be computed from $C^\ell(i,j)$ by comparing i against each possible $h_s(\ell, i+1)$. One may then compute $D(i,w)$ from $C^n(i,j)$. However, one may directly compute D from the h-indices more efficiently using an approach described in Section 1.3.3.

The v-indices can be interpreted similarly, though the ordering of $v_1 \dots v_\sigma$ is reversed. Consider $v_\sigma(\ell,j)$ for some fixed ℓ and j. This is the smallest value of i for which $C^\ell(i,j)$ does *not* exceed $C^{\ell-1}(i,j)$ by at least σ. Here, the comparison is between the WLCS of $x[1:l]$ and $y[i:j]$ and that of $x[1:l-1]$ and $y[i:j]$. The first WLCS might have more weight, and so there is unique index where the difference in weight first becomes less than σ. In this case, due to Corollary 1.3.3, the difference between $C^{\ell}(i,j)$ and $C^{\ell-1}(i,j)$ can only be less than the difference between $C^{\ell}(i-1,j)$ and $C^{\ell-1}(i-1,j)$. Now $v_\sigma(\ell,j)$ is the unique value of i after which the difference can be no more than $\sigma-1$. Similar intuition applies to all the other v-indices.

We note that the v-indices are not explicitly involved in the procedure for computing entries of D; however, they are necessary in computing the h-indices.

Recursive Computation of the Indices

We now show how to compute the h-indices $h_s(\ell,j)$ for all ℓ,j. We first show a general recursive formula for these indices, then show a more efficient strategy to compute them.

In the formula, h_s will always refer to $h_s(\ell-1,j)$ unless indices are specified. Similarly, v_s will always refer to $v_s(\ell,j-1)$ unless indices are specified.

Let $d=f(x_\ell,y_j)$, where f is the scoring function. For the general case $\ell,j>0$:

$$h_s(\ell,j)=\begin{cases}\text{if } d<s: \min\limits_{z\in[s,\sigma]}\big(\max(h_z,v_{z-(s-1)})\big)\\ \text{if } d\geq s:\\ \min\left(\min\limits_{z\in[d+1,\sigma]}\big(\max(h_z,v_{z-(s-1)})\big),v_{d-(s-1)}\right)\end{cases}\tag{1.3}$$

$$v_s(\ell,j)=\begin{cases}\text{if } d<s: \max\limits_{z\in[s,\sigma]}\big(\min(h_{z-(s-1)},v_z)\big)\\ \text{if } d\geq s:\\ \max\left(\max\limits_{z\in[d+1,\sigma]}\big(\min(h_{z-(s-1)},v_z)\big),h_{d-(s-1)}\right)\end{cases}\tag{1.4}$$

The base cases are $h_s(0,j)=j$ and $v_s(\ell,0)=0$ for all s. The first corresponds to an empty substring of x, which has an empty WLCS with any substring of y. The second corresponds to an empty substring of y, which has an empty WLCS with any substring of x. Recurrences (1.3) and (1.4) generalize the recurrences for h and v for unweighted LCS in [7], which can be recovered as a special case of our recurrence for $\sigma=1$.

We now describe the calculations for these indices, beginning with the $h_s(\ell,j)$ calculations. To calculate $h_s(\ell,j)$, we use the entries $v_s(\ell,j-1)$ and $h_s(\ell-1,j)$ (for all possible s) in addition to the value of d. It is useful to visualize the situation using Figure 1.1. In the figure, a represents the weight of the WLCS between $x[1:l-1]$ and $y[i:j-1]$ for some i. Similarly, u represents the weight of the WLCS between $x[1:l]$ and $y[i:j-1]$, and v is the WLCS between $x[1:l-1]$ and $y[i:j]$. Finally, t represents the WLCS between $x[1:l]$ and $y[i:j]$. The edges represent the relationship between the WLCS weights. First, u and v are both at least a. Further, one possible value for t could be $a+d$, since one may take the WLCS which corresponds to a and add in the match between $x[l]$ and $y[j]$, which has weight d. Alternatively, t could also be the same value as either u or v. If $t=u$, then $y[j]$ is unused in the WLCS; similarly, if $t=v$, the character $x[l]$ is unused.

The value of $h_s(\ell,j)$ has a natural interpretation: it is the first value of i for which the difference between t and u is at least s. Recall h_s will always refer to $h_s(\ell-1,j)$ unless

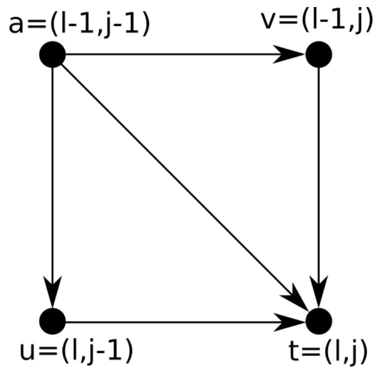

Figure 1.1: Relationship between four points.

indices are specified; similarly, v_s will always refer to $v_s(\ell, j-1)$ unless indices are specified. Thus, h_s is exactly the minimum i where there is a difference of at least s between v and a. Similarly, v_s is the minimum i where there is a difference less than s between u and a. We will relate t and u by comparing both to a. Then we can determine the correct $h_s(\ell, j)$ where $t \geq u + s$ for any $i \geq h_s(\ell, j)$.

Suppose we seek to calculate $h_s(\ell, j)$ for a fixed $s > d$. One possible weight for t is $a+d$, but this edge cannot determine h_s as $u \geq a$, and hence $a+d$ is not greater than u by at least s. The WLCS which involves a never yields any information about the minimum i where $t \geq u + s$. However, consider an i such that $i \geq h_s(\ell-1, j)$. The edge weight t can have value $a+s$. In this case, if $i \geq v_1(\ell, j-1)$, then we know that $u = a$ and hence $t \geq u + s$. Therefore, if i is greater than both h_s and v_1, then the difference between t and u is at least s. Hence, it would seem that $h_s(\ell, j)$ is just equal to $\max(h_s, v_1)$. However, there are many other *pairs* which also fulfill this condition, e.g. if i is greater than both h_{s+1} and v_2. In general, if i is greater than any h_x and $v_{x-(s-1)}$ for some positive integer $\sigma \geq x > s$, then the difference between t and u is at least s. Therefore, in the case where $s > d$ the expression for $h_s(\ell, j)$ is the minimum of the pairwise maximum of such pairs. This is formalized in equation (1.3).

The other case is when we seek to compute h_s for a fixed $s \leq d$. Here, the weight $a+d$ is always possible for t. The expression in equation (1.3) is essentially a truncated version of the expression for $s > d$. Namely, we do not need to consider the pairs involving h_s where $s \leq d$. (If $i \geq v_{d-(s-1)}$ then immediately we already know that $t \geq u + s$ regardless of whether i is also greater than some h value.)

The $v_s(\ell, j)$ computations are similar. We are interested in the difference between t and v, so we will relate both t and v to a. First, consider computing $v_s(\ell, j)$ where $s > d$. In this case we can again ignore the case where $t = a + d$. Recall that $v_s(\ell, j)$ defines the value of i where if $i > v_s(\ell, j)$, then the $t < v + s$. Consider the case where only a single important pair of values exist, h_1 and v_s. If i is greater than v_s then $t = u < a + s$. A similar property holds if $i > h_1$. Hence, $v_s(\ell, j)$ is the value of the minimum of v_s and h_1 if that is the only pair. Once again, when there are multiple pairs of v_s, h_1 and v_{s+1}, h_2 and so on, the expression becomes more complex as it becomes the maximum of the minima of these pairs.

The case for computing $v_s(\ell, j)$ when $s \leq d$ is similar to the case for the $h_s(\ell, j)$ computations in equation (1.3) except where the formula is truncated; no pairs which involve v_s for $s \leq d$ are used. Since a weight of $a + d$ can always be attained, only $h_{d-(s-1)}$ needs to be checked for any of the lesser v_s pairs.

Equations (1.3) and (1.4) give a recursive computation for all of the h-indices and v-indices. There are $O(mn\sigma)$ total entries to compute, and following the two equations above yield a $O(mn\sigma^2)$ time algorithm for computing the h- and v-indices. However, using a clever observation, it is possible to compute these entries in $O(mn\sigma)$ time, which we show next.

Faster Computation of h- and v-Indices Naively computing the recurrences 1.3 and 1.4 for each $1 \leq s \leq \sigma$ takes $O(\sigma^2)$ time. We show how to improve this to $O(\sigma)$ now.

We start with the following definitions. For $1 \leq s \leq \sigma$ define $z^*(s)$ to be the value such that the following holds: (1) $z < z^*(s) \implies v_{z-s+1} > h_z$ and (2) $z \geq z^*(s) \implies v_{z-s+1} \leq h_z$. Similarly, define $z^\#(s)$ to be the value such that (1) $z < z^\#(s) \implies h_{z-s+1} < v_z$ and (2) $z \geq z^\#(s) \implies h_{z-s+1} \geq v_z$. These values are well defined since the sequences h and v are respectively non-decreasing and non-increasing, so either the inequalities above trivially hold, or there is a point where the sequences cross. The existence of a crossing point is not affected by applying an offset to one of the sequences. We will simultaneously compute $z^*(s)$ and $z^\#(s)$ while computing new values of h_s and v_s.

To see why the above definitions are useful, consider substituting them into (1.3) and (1.4). First, consider the calculation of h_s when $d < s$:

$$\begin{aligned}
h_s(\ell, j) &= \min_{z=s}^{\sigma} (\max(h_z, v_{z-s+1})) \\
&= \min\left(\min_{z<z^*(s)} \max\left(h_z, v_{z-s+1}\right), \min_{z \geq z^*(s)} \max\left(h_z, v_{z-s+1}\right) \right) \\
&= \min\left(\min_{z<z^*(s)} v_{z-s+1}, \min_{z \geq z^*(s)} h_z \right) = \min\left(v_{z^*(s)-s}, h_{z^*(s)}\right)
\end{aligned}$$

where we again use the property that h and v are respectively non-decreasing and non-increasing. Similar calculations can be done with $z^\#(s)$ for computing $v_s(\ell, j)$ and for the case when $d \geq s$. This shows that given $z^*(s)$ and $z^\#(s)$, it is possible to compute $h_s(\ell, j)$ and $v_s(\ell, j)$ in constant time.

The only remaining task is to compute $z^*(s)$ and $z^\#(s)$ for each weight s. This can be done by sweeping through h and v in $O(\sigma)$ time. We may then compute $h_s(\ell, j)$ and $v_s(\ell, j)$ for each s in $O(\sigma)$ time.

Computing the D Matrix

We now show how to compute the entries $D(i, w)$ directly from the h-indices. The computation requires only the indices $h_s(n, j)$; in this section, we drop the n and refer to these indices simply as $h_s(j)$. We compute the entries of $D(i, w)$ row by row, iterating through one value of i at a time. At each iteration, we will keep T, a data structure storing pairs of the form $(j, h_s(j))$. During iteration i, we may insert pairs into T or delete pairs from T, maintaining the following invariant:

Algorithm 2 Reconstruct the D matrix from the h-indices

```
$T \leftarrow \emptyset$, $j \leftarrow 1$, $s \leftarrow 1$
for $i = 1, \ldots, m$ do
    while $h_s(j) \leq i$ do
        if $j > i$ then                                  ▷ Insert pairs w/ $j > i$.
            $T$.insert(($j, s, h_s(j)$))
        end if
        $s \leftarrow s + 1$
        if $s > \sigma$ then
            $s \leftarrow 1$
            $j \leftarrow j + 1$
        end if
    end while
    for $k \in K$ do                                     ▷ Compute $D(i, k)$ $\forall k$
        $(j', s', h') \leftarrow T$.search_by_rank($k$)
        $D(i, k) = j'$
    end for
    Remove from $T$ all $(j, s, h)$ where $j = i$
end for
```

$$(j, h_s(j)) \in T \iff j > i \text{ and } h_s(j) \leq i. \tag{1.5}$$

The invariant guarantees two useful properties. First, all pairs in T have $h_s(j) \leq i$, the existence of such a pair in T means that the difference between $C(i, j)$ and $C(i, j-1)$ is s. Note that if $(j, h_s(j)) \in T$, then clearly $(j, h_{s'}(j)) \in T$ for all $s' < s$ since $h_{s'}(j) \leq h_s(j)$. Thus, one can think of each pair in T as representing an increase of 1.

Second, if the pairs in T are sorted increasingly by j, then $D(i, k)$ is exactly the j_k which corresponds to the pair of rank k within T. This can be shown as follows: Let $j_1, j_2, \ldots j_k$ denote the pairs of rank 1 through k (represented by say $(j_k, h_s(j_k))$ within T. Each fixed j_x among these means that there is a difference of 1 between $C(i, j_x - 1)$ and $C(i, j_x)$. There are k pairs here, each denoting a difference of 1 between some $C(i, j_x)$ and $C(i, j_x - 1)$ and there are a total of k such differences. Note by the invariant, $j_1 > i$. Furthermore, clearly $j_1 \leq j_k$. Thus, between $C(i, i)$ and $C(i, j_k)$ there are a total of k differences of 1 each. Since $C(i, i) = 0$ the difference between $C(i, i)$ and $C(i, j_k)$ is exactly k. Hence, j_k is exactly the value of $D(i, k)$.

This analysis yields Algorithm 2, where T is a balanced tree data structure.

1.4 Analysis of Approximation and Runtime

We now formally describe the sketching strategy for the D matrix and prove the claims about the performance of Algorithm 1 under our sketching procedure.

Recall that our sketching strategy fixes a constant $\epsilon > 0$ and sets $\beta = 1 + \frac{\epsilon}{\log n}$. Define $D^*(i,s)$ to be the least j such that there exists a correspondence between x and $y[i:j]$ with weight $w \geq \lfloor \beta^s \rfloor$. Define $D^*_{r_1,r_2}$ analogously to D_{r_1,r_2} for substrings of x. To compute $D^*_{r_1,r_3}$ from $D^*_{r_1,r_2}$ and $D^*_{r_2+1,r_3}$, we modify Algorithm 1 as follows. For each power s s.t. $\lfloor \beta^s \rfloor \leq W$, we consider each power $s_1 \leq s$ and compute the least s_2 such that $\lfloor \beta^{s_1} \rfloor + \lfloor \beta^{s_2} \rfloor \geq \lfloor \beta^s \rfloor$. Let $j' = D^*_{r_1,r_2}(i, s_1)$ and $j = D^*_{r_2+1,r_3}(j'+1, s_2)$. Then there exist non-overlapping correspondences of weights at least β^{s_1} and β^{s_2}, and hence a combined correspondence of weight at least β^s, between $x[r_1:r_3]$ and $y[i:j]$. We take $D^*_{r_1,r_3}(i,s)$ to be the least j' that results from this procedure.

1.4.1 Quality of the Solution

In Section 1.3.2, we showed how to reduce the space and time requirement of computing the D matrix using Algorithm 1 via sketching. We consider only weights of the form $\lfloor \beta^s \rfloor$ for $\beta = 1 + \epsilon/\log n$ and so will not obtain the exact optimum. However, we show that we can recover a $(1-\epsilon)$ approximation to the optimum.

Computationally, we only need to construct the matrix D^* in order to extract solutions to the AWLCS and WLCS problems. However, for analysis purposes it is useful to define C^*, an approximate version of the matrix C. Let $C_{r_1,r_2}(i,j)$ be the optimal weight of a WLCS between $x[r_1:r_2]$ and $y[i:j]$. Equivalently, we have $C_{r_1,r_2}(i,j) = \max\{w \mid D_{r_1,r_2}(i,w) \leq j\}$. This motivates defining C^* as follows. Let $C^*_{r_1,r_2}(i,j) = \max_s\{\lfloor \beta^s \rfloor \mid D^*_{r_1,r_2}(i,s) \leq j\}$.

We prove the following lemma which shows that the C^* matrices approximate the C matrices well, and hence the matrices D^* implicitly encode good solutions.

Lemma 1.4.1. *Let $x[r_1, r_2]$ be a substring of x considered by our algorithm in some step. Then for all i, j we have*

$$C^*_{r_1,r_2}(i,j) \geq (1-\epsilon)\, C_{r_1,r_2}(i,j)$$

Proof. We prove the following stronger claim by induction. Let ℓ be the level at which we combined substrings to arrive at $x[r_1:r_2]$ in our algorithm. Then for all i, j we have

$$C^*_{r_1,r_2}(i,j) \geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell} C_{r_1,r_2}(i,j)$$

In particular this implies the lemma since we can use the inequality $(1-z)^\ell \geq 1 - \ell z$ for all $z \leq 1$ and $\ell \geq 0$, and the fact that $\ell \leq \log n$.

$$\begin{aligned} C^*_{r_1,r_2}(i,j) &\geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell} C_{r_1,r_2}(i,j) \\ &\geq \left(1 - \frac{\epsilon \ell}{\log n}\right) C_{r_1,r_2}(i,j) \geq (1-\epsilon) C_{r_1,r_2}(i,j) \end{aligned}$$

Now to prove the claim by induction on the levels ℓ. Fix a pair of indices i, j in y. If $x[r_1, r_2]$ was considered at the first level, then we computed the exact matrix D_{r_1,r_2} using our base case algorithm, so the base case follows trivially. Now suppose that we considered $x[r_1, r_2]$ at some level ℓ after the first. Our algorithm combines the subproblems corresponding to $x[r_1, r']$ and $x[r'+1, r_2]$ that occur at level $\ell-1$. where $r' = \lfloor (r_1+r_2)/2 \rfloor$. To compute the solution for r_1, r_2 and i, j we concatenate solutions corresponding to $x[r_1, r']$ and $y[i : j']$ and $x[r'+1, r_2]$ and $y[j'+1 : j]$, for some j'. We get the sum of their weights, but rounded down due to sketching. Thus we have:

$$C^*_{r_1,r_2}(i,j) \geq \frac{C^*_{r_1,r'}(i,j') + C^*_{r'+1,r_2}(j',j)}{(1+\epsilon/\log n)}. \tag{1.6}$$

Now $C_{r_1,r_2}(i,j) = C_{r_1,r'}(i,j'') + C_{r'+1,r_2}(j''+1,j)$ for some j'', since the solution corresponding to $C_{r_1,r_2}(i,j)$ can be written as the concatenation of two sub-solutions between $x[r_1 : r']$, $y[i : j'']$ and $x[r'+1 : r_2]$, $y[j''+1 : j]$. Note that our algorithm chooses j' such that $C^*_{r_1,r'}(i,j') + C^*_{r'+1,r_2}(j',j) \geq C^*_{r_1,r'}(i,j'') + C^*_{r'+1,r_2}(j''+1,j)$. Now applying the induction hypothesis to $C^*_{r_1,r'}(i,j'')$ and $C^*_{r'+1,r_2}(j''+1,j)$ we have:

$$\begin{aligned} C^*_{r_1,r'}(i,j') + C^*_{r'+1,r_2}(j',j) &\geq C^*_{r_1,r'}(i,j'') + C^*_{r'+1,r_2}(j''+1,j) \\ &\geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell-1} (C_{r_1,r'}(i,j'') + C_{r'+1,r_2}(j''+1,j)) \\ &= \left(1 - \frac{\epsilon}{\log n}\right)^{\ell-1} C_{r_1,r_2}(i,j) \end{aligned}$$

Now combining this with (1.6) we have:

$$C^*_{r_1,r_2}(i,j) \geq \frac{\left(1 - \frac{\epsilon}{\log n}\right)^{\ell-1} C_{r_1,r_2}(i,j)}{1 + \frac{\epsilon}{\log n}} \geq \left(1 - \frac{\epsilon}{\log n}\right)^{\ell} C_{r_1,r_2}(i,j),$$

which completes the proof of the general case. □

1.4.2 Running Time

We now analyze the running time of our algorithm, starting with the combining procedure in Algorithm 1.

Lemma 1.4.2. *Let $n' = r_3 - r_1 + 1$ be the number of characters of the string x assigned to one call to Algorithm 1. Let $\epsilon' = \epsilon/2\log(n)$ and $t = \log_{1+\epsilon'}(\sigma \min(n', m))$. Then the procedure described in Algorithm 1 uses $O(mt)$ space and runs in $O(mt^2)$ time.*

Proof. First note that $\sigma \min(n', m)$ is an upper bound on the maximum weight for the instance passed to Algorithm 1. The matrix D contains an entry for each pair of starting indices and each weight. There are m starting indices. To prove the space bound, it suffices to show that the number of possible weights is $O(t)$. Recall that sketching the weights yields an entry for each weight of the form $(1+\epsilon/2\log_2 n)^\ell = (1+\epsilon')^\ell$ for integer ℓ, up to some upper bound on the weight. Hence, taking t as in the statement of the lemma implies that $(1+\epsilon')^t \geq \sigma \min(n', m)$, so $O(t)$ different weights suffices.

The bound on the running time follows by noting that each of the $O(m)$ iterations of the algorithm iterates through all pairs of weights, yielding total time $O(mt^2)$. □

$O(\log(p))$ rounds of combining are required to merge the p base-case results into the final D matrix, since each round reduces the number of remaining sub-problems by half. To analyze the entire algorithm, we separately consider the two steps.

Lemma 1.4.3. *Let $B(m, n)$ be the running time of a base-case algorithm computing D. Then our algorithm runs in time $B(m, n/p) + O(m \log^2_{1+\epsilon'}(\sigma m) \log(p))$ on p processors.*

Proof. The base case algorithm is run on subproblems of size $n/p \times m$. Each of the $O(\log(p))$ rounds of merging has cost $O(m \log^2_{1+\epsilon'}(\sigma m))$ by the previous lemma. □

Finally, since $\log_{1+\epsilon'}(\sigma m) = O(\log(\sigma m)/\epsilon') = O(\log_2(n)\log(\sigma m)/\epsilon)$, our algorithm acheives the claimed runtime and local memory per processor.

Chapter 2

Distributed Algorithms for Hierarchical Clustering

This chapter is based on "A Framework for Parallelizing Hierarchical Clustering Methods" [99], which appeared in the proceedings of European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD) 2019.

2.1 Introduction

Thanks to its ability in explaining nested structures in real world data, hierarchical clustering is a fundamental tool in any machine learning or data mining library. In recent years the method has received a lot of attention, with some work developing new foundations and objective functions for hierarchical clustering [53, 130, 45, 146]. Other work has looked at scaling specific methods such as single-linkage [87, 34]. Despite these efforts, almost all proposed hierarchical clustering techniques are sequential methods that are difficult to apply to large data sets.

The input to the hierarchical clustering problem is a set of points and a function specifying either their pairwise similarity or their dissimilarity. The output of the problem is a rooted tree representing a hierarchical structure of the input data, also known as a dendrogram. The input points are the leaves of this tree and subtrees induced by non-leaf nodes represent clusters. These clusters should also become more refined when the root of the corresponding subtree is at a lower level in the tree. Hierarchical clustering is useful because the number of clusters does not need to be specified in advance and because the hierarchical structure yields a taxonomy that allows for interesting interpretations of the data set. For an overview of hierarchical clustering methods refer to [118, 95, 77].

Several algorithms have emerged as popular approaches for hierarchical clustering. Different techniques are used depending on the context because each method has its own advantages and disadvantages. There are various classes of data sets where each method outperforms the others. For example, the centroid-linkage algorithm has been used for biological data such as genes [63], whereas, an alternative method, bisecting k-means, is popular for document comparison [139]. The most commonly used methods can be

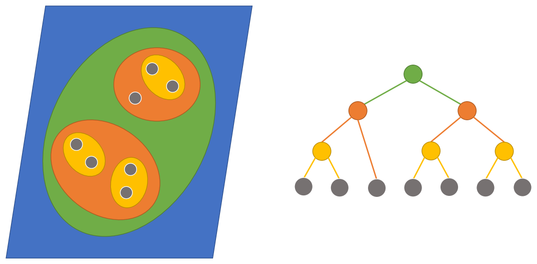

Figure 2.1: A Hierarchical Clustering Tree. The grey leaves are the input data points. Internal nodes represent a cluster of the leaves in the subtree rooted at the internal node.

categorized into two families: agglomerative algorithms and divisive algorithms.

Divisive algorithms are top-down. They partition the data starting from a single cluster and then refine the clusters iteratively layer by layer. The most commonly used techniques to refine clusters are k-means, k-median, or k-center clustering with $k = 2$. These divisive algorithms are known as bisecting k-means (respectfully, median, center) algorithms [83]. Agglomerative algorithms are based on a bottom up approach (see [76] for details). In an agglomerative algorithm, all points begin as their own cluster. Clusters are then merged through some merging strategy. The choice of merging strategy determines the algorithm. Common strategies include single-linkage, average-linkage and centroid-linkage.

Most of these algorithms are inherently sequential; they possess a large number of serial dependencies and do not lend themselves to efficient parallelization. For example, in a divisive method lower splits in the tree depend upon splits higher up in the tree, so if the depth of the resulting tree is large then there will be a large number of serial dependencies.

Recently, there has been interest in making hierarchical clustering scalable. Nevertheless most prior work has focused on scaling the single-linkage algorithm; efficient MapReduce and Spark algorithms are known for this problem [86, 87, 34, 153]. This is unsurprising because single-linkage can be reduced to computing a Minimum-Spanning-Tree [74], and there has been a line of work on efficiently computing minimum spanning trees in parallel and distributed settings [23, 90, 101, 12]. Unfortunately this approach does not extend to other hierarchical clustering techniques such as divisive methods.

Contributions: In this chapter we introduce scalable hierarchical clustering algorithms. The main results are the following:

A Theoretical Framework: We develop a theoretical framework for scaling hierarchical clustering methods. We introduce the notion of *closeness* between two hierarchical clustering algorithms. Intuitively, two algorithms are close if they make provably close or similar decisions. This enforces that our scalable algorithms produce similar solutions to their sequential counterpart. Using this framework, we formalize the root question for scaling existing methods.

Provably Scalable Algorithms: We introduce fast scalable algorithms for the bisecting k-means, k-median and k-center algorithms. These new algorithms are the main contribution of the chapter. The algorithms are proved to be close to their sequential counterparts and efficient in parallel and distributed models. These are the first scalable algorithms for divisive k-clustering, moreover they are applicable to data belonging to any metric space.

2.2 Preliminaries

In this section we formally define the hierarchical clustering problem, describe popular sequential approaches, and provide other necessary background information.

Problem Input: The input is a set S of n data points. The distance between points specifies their dissimilarity. Let $d(u, v) \geq 0$ denote the distance between two points $u, v \in S$. It is assumed that d is a metric.

Problem Output: The output is a rooted tree where all of the *input* points are at the leaves. Internal nodes represent clusters; the leaves of the subtree rooted at a node correspond to the points in that specific cluster.

Computational Model: We will analyze our algorithms in the context of the MPC model as described in the Introduction to this dissertation. Here the input size N is given by the number of data points n and we assume that the distance function d is given by an oracle which we can query for any two points on the same machine.

k-Clustering Methods: We recall the definitions of k-$\{$center,median,means$\}$ clusterings. Let $C = \{c_1, c_2, \ldots, c_k\}$ be k distinct points of S called centers. For $x \in S$ let $d(x, C) = \min_{c \in C} d(x, c)$ We say that these centers solve the k-$\{$center,median,means$\}$ problem if they optimize the following objectives, respectively:

- k-center: $\min_C \max_{x \in S} d(x, C)$
- k-median: $\min_C \sum_{x \in S} d(x, C)$
- k-means: $\min_C \sum_{x \in S} d(x, C)^2$.

The choice of centers induces a clustering of S in the following natural way. For $i = 1, \ldots, k$ let $S_i = \{x \in S \mid d(x, c_i) = d(x, C)\}$, that is we map each point to its closest center and take the clustering that results. In general it is NP-hard to find the optimal set of centers for each of these objectives, but efficient $O(1)$-approximations are known [78, 47, 89].

Classic Divisive Methods: We can now describe the classical divisive k-clustering algorithms. The pseudocode for this class of methods is given in Algorithm 3. As stated before, these methods begin at the root of the cluster tree corresponding to the entire set S and recursively partition the set until we reach the leaves of the tree. Note that at each node of the tree, we use an optimal 2-clustering of the current set of points to determine the two child subtrees of the current node.
Notation: We present some additional notation and a few technical assumptions. Let X be a finite set of points and x a point in X. We define the ball of radius R around x, with notation $B(x, R)$, as the set of points with distance at most R from x in the point set X, i.e. $B(x, R) = \{y \mid d(x, y) \leq R, y \in X\}$. Let $\Delta(X) = \max_{x,y \in X} d(x, y)$ be the maximum distance between points of X. Finally, WLOG we assume that all points and pairwise

Algorithm 3 DivisiveClustering(S)

```
1: if $|S| = 1$ then
2:     Return a leaf node corresponding to $S$
3: else
4:     Let $S_1, S_2$ be an optimal 2-clustering of $S$  ▷ One of the means, median, or center objective is used
5:     $T_1 \leftarrow$ DivisiveClustering($S_1$)
6:     $T_2 \leftarrow$ DivisiveClustering($S_2$)
7:     Return a tree with root node $S$ and children $T_1$, $T_2$
8: end if
```

distances are distinct[1] and that the ratio between the maximum and minimum distance between two points is polynomial in n.

2.3 A Framework for Parallelizing Hierarchical Clustering Algorithms

We now introduce our theoretical framework that we use to design and analyze scalable hierachical clustering algorithms. Notice that both divisive and agglomerative methods use some cost function on pairs of clusters to guide the decisions of the algorithm. More precisely, in divisive algorithms the current set of points S is partitioned into S_1, and S_2 according to some cost $c(S_1, S_2)$. Similarly, agglomerative algorithms merge clusters S_1 and S_2 by considering some cost $c(S_1, S_2)$. So in both settings the main step consists of determining the two sets S_1 and S_2 using different cost functions. As an example, observe that $c(S_1, S_2)$ is the 2-clustering cost of the sets S_1 and S_2 in the divisive method above and that $c(S_1, S_2) = d(\mu(S_1), \mu(S_2))$ in centroid linkage.

The insistence on choosing S_1, S_2 to minimize the cost S_1, S_2 leads to the large number of serial dependencies that make parallelization of these methods difficult. Thus, the main idea in this chapter is to obtain more scalable algorithms by relaxing this decision making process to make near optimal decisions. This is formalized in the following definitions.

Definition 2.3.1 (α-close sets)**.** *Let c be the cost function on pairs of sets and let S_1, S_2 be the two sets that minimize $c(S_1, S_2)$. Then we say that two sets S_1', S_2' are α-**close** to the optimum sets for cost c if $c(S_1', S_2') \leq \alpha c(S_1, S_2)$, for $\alpha \geq 1$.*

Definition 2.3.2 (α-close algorithm)**.** *We say that a hierarchical clustering algorithm is α-**close** to the optimal algorithm for cost function c if at any step of the algorithm the sets selected by the algorithm are α-**close** for cost c, for $\alpha \geq 1$.* [2]

[1]We can remove this assumption by adding a small perturbation to every point.

[2]Note that the guarantees is on each single choice made by the algorithm but not on all the choices together.

A necessary condition for efficiently parallelizing an algorithm is that it must not have long chains of dependencies. Now we formalize the concept of chains of dependencies.

Definition 2.3.3 (Chain of dependency)**.** *We say that a hierarchical clustering algorithm has a chain of dependencies of length ℓ, if every decision made by the algorithm depends on a chain of at most ℓ previous decisions.*

We now define the main problem tackled in this chapter.

Problem 1. *Is it possible to design hierarchical clustering algorithms that have chain of dependencies of length at most* $\mathrm{poly}(\log n)$ *and that are α-close, for small α, for the k-means, k-median, and the k-center cost functions?*

It is not immediately obvious that allowing our algorithms to be α-close will admit algorithms with small chains of dependencies. In Section 2.4.1 we answer this question affirmatively for divisive k-clustering methods [3]. Then in section 2.4.2 we show how to efficiently implement these algorithms in the MPC model.

2.4 Algorithms and Theoretical Guarantees

2.4.1 Distributed Divisive k-Clustering

We now present an $O(1)$-close algorithm with dependency chains of length $O(\log(n))$ under the assumption that the ratio of the maximum to the minimum distance between points is polynomial in n.

As discussed in Sections 2.2 and 2.3, the main drawback of Algorithm 3 is that its longest chains of dependencies an be linear in the size of the input[4]. We modify this algorithm to overcome this limitation while remaining $O(1)$-close with respect to the clustering cost objective. In order to accomplish this we maintain the following invariant. Each time we split S into S_1 and S_2, each set either contains a constant factor fewer points than S or the maximum distance between any two points has been decreased by a constant factor compared to the maximum distance in S. This property will ensure that the algorithm has a chain of dependency of logarithmic depth. We present the pseudocode for the new algorithm in Algorithms 4 and 5.

The goal of this subsection is to show the following theorem guaranteeing that Algorithm 4 is provably close to standard divisive k-clustering algorithms, while having a small chain of serial dependencies.

Theorem 2.4.1. *Algorithm 4 is $O(1)$-close for the k-center, k-median, and k-means cost functions and has a chain of dependencies of length at most $O(\log n)$.*

[3]In prior work, Yaroslavtsev and Vadapalli [153] give an algorithm for single-linkage clustering with constant-dimensional Euclidean input that fits within our framework.

[4]Consider an example where the optimal 2-clustering separates only 1 point at a time.

The main difference between Algorithm 3 and Algorithm 4 is the reassignment step. This step's purpose is to ensure that the invariant holds at any point during the algorithm as shown in the following lemma. Intuitively, if both S_1 and S_2 are contained within a small ball around their cluster centers, then the invariant is maintained. However, if this is not the case, then there are many points "near the border" of the two clusters, so we move these around to maintain the invariant.

Lemma 2.4.2. *After the execution of Reassign(S_1, S_2) in Algorithm 4 either $|S_1| \leq \frac{7}{8}|S|$ or $\Delta(S_1) \leq \frac{1}{2}\Delta(S)$. Similarly, either $|S_2| \leq \frac{7}{8}|S|$ or $\Delta(S_2) \leq \frac{1}{2}\Delta(S)$.*

Proof. Let S_1, S_2 be the resulting clusters and v_1, v_2 be their centers. Consider the sets $B_i = B(v_i, \Delta(S)/8) \cap S$, for $i = 1, 2$. If $S_1 \subseteq B_1$ and $S_2 \subseteq B_2$, then both clusters are contained in a ball of radius $\Delta(S)/8$. By the triangle inequality the maximum distance between any two points in either S_1 or S_2 is at most $\Delta(S)/2$.[5]

Otherwise, consider U, the set of points not assigned to any B_i. We consider $c = 4$ as in the default case. If $|U| \leq |S|/4$, then the algorithm assigns U to the smaller of B_1 and B_2 and the resulting cluster will have size at most $3|S|/4$ since the smaller set has size at most $|S|/2$. Furthermore the other cluster is still contained within a ball of radius $\Delta/8$ and thus the maximum distance between points is at most $\Delta(S)/2$. If $|U| \geq |S|/4$ then the points in U are distributed evenly between S_1 and S_2. Both sets in the recursive calls are guaranteed to have less than $|S| - |U|/2 \leq \frac{7}{8}|S|$ points since U was evenly split. Similar properties can be shown for other values of c. □

Next we can show that the algorithm is $O(1)$-close by showing that the algorithm is an $O(1)$-approximation for the desired k-clustering objective in each step.

Lemma 2.4.3. *Let p be the norm of the clustering objective desired (i.e. $p = 2$ for k-means, $p = \infty$ for k-center or $p = 2$ for k-median). The clustering produced in each iteration is a constant approximation to any desired k-clustering objective with any constant norm $p \in (0, 2]$ or $p = \infty$.*

Proof. Fix p to be a constant in $(0, 2]$ or ∞ and let $k = 2$ be the desired number of centers. Consider a two clustering solution with centers v_1 and v_2 that is a c-approximation. The proof considers the cases for p constant or $p = \infty$ separately.

For p constant. For any two centers v_1 and v_2 the cost of the solution is $\sum_{u \in S_1} d(u, v_1)^p + \sum_{u \in S_2} d(u, v_2)^p$. So by definition of U, this is at least

$$O := \sum_{u \in S_1 \setminus U} d(u, v_1)^p + \sum_{u \in S_2 \setminus U} d(u, v_2)^p + |U|(\Delta(S)/8)^p.$$

Now the algorithm reassigns points in U and the maximum distance between any point and its new center is upper-bounded by $\Delta(S)$. So the cost of the algorithm's solution is at most $\sum_{u \in S_1 \setminus U} d(u, v_1)^p + \sum_{u \in S_2 \setminus U} d(u, v_2)^p + |U|(\Delta(S))^p \leq 4^p O$. Thus the solution

[5] By the generalized triangle inequality this is true for $p = 1, 2$ and it is true for $p = \infty$. So this is true for the cost of k-center, k-means and k-median.

computed by Algorithm 4 is at most a factor 8^p larger due to the reassignment and so it is a $8^p c$-approximation if we used a c-approximation algorithm to solve the clustering problem.

For $p = \infty$. If $|U|$ is empty the algorithm returns the c-approximation, otherwise the cost of the c-approximation algorithm is at least $\Delta(S)/8$ and our algorithm returns a solution of cost at most $\Delta(S)$. Thus, the algorithm computes an $8c$-approximation giving the lemma. □

Combining Lemma 2.4.2 and Lemma 2.4.3 we obtain Theorem 2.4.1 as a corollary.

2.4.2 From Bounded Length Dependency Chains to Parallel Algorithms

We now discuss how to adapt our algorithms to run on distributed systems. In particular we show that every iteration between consequent recursive calls of our algorithms can be implemented using a small number of rounds in the massively parallel model of computation and so we obtain that both algorithms can be simulated in a polylogarithmic number of rounds.

Parallelizing Divisive k-Clustering: We start by observing that there are previously known procedures [64, 46, 35] to compute approximate k-clusterings in the massively parallel model of computation using only a constant number of rounds. Here we use these procedures as a black-box.

Next, the reassignment operation can be performed within a constant number of parallel rounds. Elements can be distributed across machines and the centers v_1 and v_2 can be sent to every machine. In a single round, every element computes the distance to v_1 and v_2 and in the next round the size of B_1, B_2 and U are computed. Finally given the sizes of B_1, B_2 and U the reassignment can be computed in a single parallel round.

So steps 4, 5 and 6 of Algorithm 4 can be implemented in parallel using a constant number of parallel rounds. Furthermore, we established that the algorithm has at most logarithmic chain of dependencies. Thus we obtain the following theorem:

Theorem 2.4.4. *There exist $O(\log n)$ round hierarchical clustering algorithms in the MPC setting that are $O(1)$-close to divisive k-means, divisive k-center, and divisive k-median.*

Algorithm 4 $O(1)$-close divisive k-clustering algorithm

```
1: if $|S| = 1$ then
2:     Return a leaf node corresponding to $S$
3: else
4:     Let $S_1, S_2$ be an optimal 2-clustering of $S$ ▷ One of the means, median, or center objective is used
5:     $(S_1, S_2) \leftarrow$ Reassign$(S_1, S_2, \Delta(S))$
6:     Recurse on each of $S_1$ and $S_2$, yielding trees $T_1, T_2$
7:     Return a tree with root $S$ and children $T_1, T_2$
8: end if
```

Algorithm 5 Reassign(S_1, S_2, Δ)

```
1: Let $v_1, v_2$ be the centers of each cluster
2: for $i = 1, 2$ do
3:     $B_i \leftarrow B(v_i, \Delta/8) \cap (S_1 \cup S_2)$ ▷ Construct a ball of radius $\Delta/4$ around each cluster center
4: end for
5: if $S_1 \subseteq B_1$ and $S_2 \subseteq B_2$ then
6:     Return $(S_1, S_2)$
7: else
8:     $U \leftarrow (S_1 \setminus B_1) \cup (S_2 \setminus B_2)$    ▷ $U$ is the set of all points not assigned to any ball]
9:     if $|U| \leq n/c$ then
10:        $\{c$ is a constant parameter. By default, $c = 4\}$
11:        Assign $U$ to the smaller of $B_1$ and $B_2$.
12:    else
13:        Split $U$ evenly between $B_1$ and $B_2$
14:    end if
15:    Return $(B_1, B_2)$
16: end if
```

Chapter 3

Online Load Balancing with Predictions

This chapter is primarily based on the paper "Online Scheduling via Learned Weights" [100], which appeared in the proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA) 2020. Collaborators on the project were Silvio Lattanzi, Benjamin Moseley, and Sergei Vassilvitskii. We also include a result that appeared in "Learnable and Instance Robust Predictions for Online Matching, Flows, and Load Balancing" [103], as it complements the results which appeared in the preceding paper. Collaborators on the latter project were R. Ravi, Benjamin Moseley, and Chenyang Xu.

3.1 Introduction

Modern machine learning has had unprecedented success in speech and language understanding, vision, and perception. More recently, researchers have shown how to use machine learning to solve classical combinatorial optimization questions, for example finding the optimal decision strategy for the secretary problem [93], computing Optimal Auctions [61], or building faster look up tables [94].

The above approaches achieve notable empirical success, but come without any theoretical analysis. A complementary line of work has looked into ways to incorporate machine learned predictions to provide formal guarantees, for instance improving competitive ratios for online algorithms [108, 126]. These methods bound the performance of the algorithm in terms of the (ex-post) error of the predictor, and show that, with good, but *imperfect*, predictions, one can circumvent strong worst case lower bounds.

In this work we continue this line of research and extend it to one of the central scheduling problems: minimizing makespan under restricted assignment. In this problem there are m machines, and jobs arrive one at a time, each annotated with its size and the subset of machines it can run on. The goal is to allocate jobs to machines online, to minimize the *makespan*, i.e. the maximum total size of the jobs on any machine. This is a key problem in scheduling, but has strong, $\Omega(\log m)$, lower bounds on the competitive ratio.

Online algorithms operate under worst case assumptions about the input, but, in practice, the input often has a repetitive nature to it. As an example, when scheduling jobs in a large data center, we may observe that some jobs happen daily around the same time (for instance, regular backup jobs), and so their presence is very predictable. Other jobs, for instance, payroll analysis jobs, may have some variability in terms of arrival time, but are known to happen weekly, or at the end of each month; overall traffic may be lighter on holidays, or during extreme weather events, and so on. Thus, it makes sense that we can predict something about the jobs and the congestion of machines using standard features, such as day of week, weekday vs. holiday, weather, etc.

While the predictions are not going to be perfect, they can still guide the allocation algorithm about which machines will be in contention in the future. The nature of the prediction then becomes part of the algorithm design process. There is a fine balance between offloading too much of the complexity onto the predictor on one side, versus having the prediction not provide any insights on the other. The former makes the algorithmic problem trivial, but the learning problem virtually impossible, the latter gives little additional benefit over worst case analysis. Thus, we strive for sparse representations that capture the complexity of the problem. For instance, for online bipartite matching, a problem closely related to our work (except for the choice of the objective function), previous work [54, 145] has shown that the dual variables associated with machines in the LP formulation of the problem are enough to reconstruct the optimum assignment. As we will show in Section 3.2.3 this representation is not sufficient when minimizing makespan, nor are other immediate quantities, such as the total load of each machine in the optimum solution, or the number of jobs that could potentially be assigned.

Our first contribution is in identifying a specific quantity that we will predict. We follow the work of [5] and associate a *weight*, w_i with each machine i. We show that even if we are only given access to estimates, $\hat{w}_i$ of the optimal w_i, we can recover a near optimal fractional solution, with the approximation guarantee that scales logarithmically with the error.

Theorem 3.1.1 (Theorem 3.3.3 restated)**.** *Let w be a set of machine weights that lead to a fractional solution with makespan T. Let $\hat{w}$ be predictions of w and let $\eta = \max_i \hat{w}_i / w_i$ be the maximum error in our predictions. Then there exists an online algorithm (which is given T as an input) that generates a fractional assignment of jobs to machines with makespan at most $O(T \log \eta)$.*

We note that this result assumes that the algorithm knows the value of T. Later in Section 3.9, we show how to remove this assumption, at a small cost of a factor of $O(\log \log \log(m))$ to the makespan of the resulting solution.

The learned weights allow us to recover an approximately optimal *fractional* solution; however, new algorithmic challenges arise when rounding this solution online to an *integral* solution. Our second technical contribution is an online rounding procedure that loses an $O((\log \log m)^3)$ factor in the makespan. Thus converting from fractional to integer solutions is not as hard as obtaining good fractional solutions in the first place. We note that our rounding procedure can be used for the more general unrelated machines problem, where job sizes are machine dependent and uncorrelated.

Rounding a fractional solution online requires several new ideas. Prior rounding algorithms need access to the overall instance, and are inherently offline. For instance, the 2-approximation of Lenstra, Shmoys and Tardos [105] requires preprocessing the fractional solution, iteratively transforming the assignment by shifting probability mass found in cycles in the corresponding bipartite graph. Obviously the full problem instance and assignment is necessary to perform this kind of rebalancing. Other methods utilize sophisticated configuration LPs[84, 140], which cannot be easily modified to fit the online setting. Thus known makespan rounding methods are not good candidates to adapt to the online setting.

In an effort to design a good rounding algorithm, we first observe that any deterministic rounding procedure has a competitive ratio of $\Omega(\log m)$ as compared to the fractional solution (See Section 3.6.1). We resort to randomized approaches. Since simple randomized rounding will lead to a poor approximation ratio, we introduce a new rounding algorithm that involves carefully transforming the instance to ensure certain structural properties, then classifying jobs into different categories, and running different rounding algorithms for each category. The classification is critical, as rounding algorithms that are good for one category perform poorly for others. However, coupled together in the right way, we establish our results. See Section 3.4 for a proof overview and Section 3.5 for the whole proof.

Theorem 3.1.2 (Theorem 3.5.1 restated)**.** *Let x be a fractional assignment of restricted assignment jobs that is revealed online and let T be the fractional makespan of x.*[1] *There exists a randomized online algorithm that rounds a fractional assignment to an integer assignment such that the resulting makespan is at most $O((\log\log m)^3 T)$ with high probability.*

In addition to this, we show that our online rounding algorithm is *nearly optimal.* We show that no randomized online rounding algorithm can achieve a competitive ratio better than $\tilde{\Omega}(\log\log m)$ when rounding a given fractional solution online (See Section 3.6.2).

Theorem 3.1.3 (Theorem 3.6.1 restated)**.** *Let x be a fractional assignment of restricted assignment jobs that is received online and let T be the fractional makespan. No deterministic algorithm for converting x to an integer assignment can be $o(\log / \log\log mm)$-competitive with respect to T. Further, no randomized algorithm for the same task can be $o(\log\log m / \log\log\log m)$-competitive with respect to T.*

Combining the two upper bound results, we show that by learning approximate weights and applying our rounding algorithm one obtains an $O(\log \eta (\log\log m)^3)$-competitive algorithm, an exponential improvement over an algorithm that does not use any predictions, whenever the predictions are reasonably accurate. We also show that it's easy to fall back to the traditional algorithm in case when the error is detected to be large. More formally:

Corollary 3.1.4. *For any restricted assignment job sequence there exist machine weights w and an online algorithm that when given access to predictions $\hat{w}$ of the weights, assigns the*

[1]We assume that T is at least the size of any job to deal with instances having poor integrality gap. See Section 3.5 for a proper definition of T

jobs with competitive ratio $\tilde{O}(\min\{\log \eta (\log\log m)^3, \log m\})$ *with respect to the makespan, where* $\eta = \max_i \hat{w}_i / w_i$. *The online algorithm is randomized and succeeds with high probability.*

Finally, we show that the weights can in some sense be learned from past problem instances. The next result is concerned with a PAC-style model.

Theorem 3.1.5 (Theorem 3.6.9 restated)**.** *Let* $\epsilon, \delta \in (0,1)$ *and* $R = O(\frac{m^2}{\epsilon^2}\log(\frac{m}{\epsilon}))$ *be given and let* $\mathcal{D} = \prod_{j=1}^n \mathcal{D}_j$ *be a distribution over n-job restricted assignment instances such that* $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{OPT}(S)] \geq \Omega(\frac{1}{\epsilon^2}\log(\frac{m}{\epsilon}))$. *There exists an algorithm which finds weights* $w \in \mathcal{W}(R)$ *such that* $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w,S)] \leq (1+O(\epsilon))\min_{w'\in\mathcal{W}(R)}\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w',S)]$ *when given access to* $s = \mathrm{poly}(m, \frac{1}{\epsilon}, \frac{1}{\delta})$ *independent samples* $S_1, S_2, \ldots, S_s \sim \mathcal{D}$. *The algorithm succeeds with probability at least* $1 - O(\delta)$ *over the random choice of samples.*

3.1.1 Related Work

There are multiple instances of augmenting classic online algorithms with additional information in order to improve competitive ratios. Broadly, the extra information can come in the form of assumptions on the data, or in terms of explicit hints about the future given to the algorithm.

The most prominent example of the former is the random arrival model, where the full set of arrivals is chosen adversarially, but the exact sequence seen by the algorithm is a random permutation of the worst case instance. These lead to a family of algorithms that use the first part of the input to learn the structure of the input, often via simple empirical risk minimization (ERM) methods, and then use the learned structure to guide the algorithm's choices on the rest of the input. Notable examples include the secretary problem, where the learned structure is just the best candidate seen so far, to more delicate arguments, such as those in the work by Devanur and Hayes [54] and Vee et al. [145] for online bipartite matching. A line of work initiated by Cole and Roughgarden [50] and continued by Balkanski et al. [30] asks explicitly what kind of information can be learned just from a sample of the data, and strives to get tight bounds on the size of the sample necessary to achieve good results.

A related structural assumption is that input comes from a stochastic distribution. This too has been studied in the context of online matching [113] and bandit learning [41], where the authors show how to get improved guarantees if the assumption holds, and retain the worst case guarantees if it does not. Mahdian et al. [109] generalize this even further, allowing for an arbitrary *optimistic* algorithm, rather than assuming that the input is stochastic, and showing how to recover a constant fraction of either the optimistic or worst case performance.

In a new line of work, Kumar et al. [98] assume that the online input is a mixture of adversarial elements and elements arriving from a known sequence. They show how to achieve near optimal results for the online matching in this "semi-online" model of computation.

The work described above assumes that the input is restricted in some manner. Additionally, there has been increased interest in a model, where the input is not explicitly restricted, but some information about it is available to the algorithm. The seminal work by Ailon et al. [6] considered "self-improving" algorithms that effectively learn the input distribution, and adapt to be nearly optimal in that domain.

In the online algorithms with advice model, the algorithm has access to an oracle that knows the exact input. The goal then is to reduce the amount of information the oracle needs to communicate to the algorithm, see Boyar et al. [38] for a recent survey. Critically, in this model the oracle is perfect, and makes no mistakes, whereas we explicitly assume that the predictions are going to be error-prone, and our goal is to tie the competitive ratio of the algorithm to the quality of the predictions.

Another closely related area of work is that of approximation and online algorithms for scheduling. There has been a considerable amount of work on approximate makespan minimization. The work of Lenstra, Shmoys and Tardos [105] gives a 2-approximation algorithm by rounding a natural LP relaxation and this result holds for the unrelated machines case. The breakthrough work of [84, 140] improves the approximation ratio for the restricted assignment problem. The work of Svensson [140] shows a 1.9412 approximation and Jansen and Rohwedder [84] improves this to a $2-\frac{1}{6}+\epsilon$ approximation. This additionally bounds the integrality gap of the configuration LP by $2-\frac{1}{6}$. For the unrelated machine case, the best known approximation is 2. Recently, Chakrabarty, Khanna and Li [44] show improved results for a special case. In the online setting there are tight bounds of $\Theta(\log m)$ on the competitive ratio for the restricted assignment problem [21].

Finally, we refer the reader back to the introduction of this dissertation for a review of recent work in algorithms with predictions and data-driven algorithm design.

3.2 Preliminaries

3.2.1 Problem Definition and Notation

We study the online restricted assignment problem. In this problem there are m machines (indexed from 1 to m) and a sequence of jobs that arrive online. Job j has an integer size $p_j > 0$ and a subset of machines $\emptyset \subsetneq N(j) \subseteq [m]$ where j can be feasibly assigned. Throughout this chapter we refer to $N(j)$ as the *neighborhood* of job j. Similarly, we can define the neighborhood of a machine i, $N(i)$ as the set of jobs that can feasibly be assigned to i. The neighborhood structure induces a bipartite graph on the set of machines and the set of jobs. The algorithm must irrevocably assign each job j to some machine $i \in N(j)$, and it must do so online, i.e. with no knowledge of the future jobs in the sequence. This assignment incurs a load on each machine, equal to the total size of the jobs assigned to each machine. The objective is to minimize the makespan, or maximum load across all machines. If J_i is the set of jobs assigned to machine i, then $L_i = \sum_{j \in J_i} p_j$ and the makespan, $T = \max_{i \in [m]} L_i$.

A more general version of this problem is the online makespan minimization on unrelated machines problem. The setup for this problem is mostly the same as above, except that instead of jobs having a single size and a subset of feasible machines, the size of a job is completely machine dependent and uncorrelated between machines. That is, for each machine-job pair i, j, there is a size $p_{ij} > 0$ that determines the size of job j if it were assigned to machine i. The objective is still to minimize the makespan. The restricted assignment problem can be reduced to unrelated machines by taking $p_{ij} = p_j$ if $i \in N(j)$ and $p_{ij} = \infty$ otherwise.

We study our algorithms in the setting of competitive analysis. If T is the optimal makespan for a sequence of jobs in hindsight, then we seek to give online algorithms that output assignments with makespan at most αT, for $\alpha \geq 1$ as small as possible. If an online algorithm achieves such a guarantee, then we say that the algorithm is α-competitive. For the case of the online rounding problem we define competitiveness similarly, but instead with $T = \max\{\max_{i\in[m]} \sum_{j\in N(i)} p_j x_{ij}, \max_j p_j\}$. In the case of randomized algorithms, we compare our online algorithm to oblivious adversaries. The adversary does not have access to the randomness used by our algorithm to make assignments. That is, the adversary fixes the (worst case) sequence of jobs in the beginning, then our algorithm runs and assigns the jobs.

In analyzing randomized algorithms we will often say that some event occurs with high probability. We take this to mean that the event occurs with probability at least $1 - 1/m^c$, for any constant $c > 0$. This implies that we can union bound over any polynomially in m many "bad" events and retain high probability. We will use concentration inequalities to establish high probability bounds several times in our analysis, see below for precise statements.

Theorem 3.2.1 (Upper Chernoff Bound). *Let $X_1, X_2, \ldots, X_n$ be independent random variables with $X_i \in [0, 1]$ for $1 \leq i \leq n$. Let $X = \sum_i X_i$ and $\mu \geq E[X]$, then for all $\epsilon > 0$ we have*

$$\Pr[X \geq (1+\epsilon)\mu] \leq \exp\left(-\frac{\epsilon^2}{2+\epsilon}\mu\right)$$

Theorem 3.2.2 (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 $\mu = \sum_i \mathbb{E}[X_i]$. Then for all $\delta \in (0, 1)$*

$$\Pr[X < (1-\delta)\mu] \leq \exp\left(\frac{-\delta^2\mu}{2}\right).$$

Theorem 3.2.3 (Bernstein's Inequality). *Let $X_1, \ldots, X_t$ be independent Bernoulli RV's with $\Pr[X_i = 1] = p_i$ and let $a_1, \ldots, a_t$ be non-negative scalars. Let $X = \sum_i a_i X_i$, $v = \sum_i a_i^2 p_i$, and $a = \max_i a_i$. Then for all $\lambda > 0$*

$$\Pr[X > \mathbb{E}[X] + \lambda] \leq \exp\left(\frac{-\lambda^2}{2v + 2a\lambda/3}\right).$$

Theorem 3.2.4 (Two-Sided Hoeffding Bound). *Let $X_1, X_2, \ldots, X_n$ be independent random variables with $a_i \leq X_i \leq b_i$ for $1 \leq i \leq n$. Let $X = \sum_i X_i$ and $\mu = \mathbb{E}[X]$. Then for all $t > 0$ we have*

$$\Pr[|X - \mu| \geq t] \leq 2\exp\left(-\frac{t^2}{\sum_i (b_i - a_i)^2}\right)$$

Theorem 3.2.5 (Union Bound). *Let $A_1, \ldots, A_t$ be a collection of events, then*

$$\Pr[A_1 \cup \ldots \cup A_t] \le \sum_{i=1}^{t} \Pr[A_i]$$

Technical Assumptions. Throughout the rest of this chapter, we assume that our algorithms know the optimal makespan T. In the offline setting this assumption can typically be removed by a simple binary search. In Section 3.9, we show how to remove this assumption in the online setting. We also assume that the job sizes are at least 1 and are polynomially bounded, i.e. $p_j = O(m^k)$ for some constant k. This assumption can be made with negligible increase in the competitive ratio.

3.2.2 ML Oracles

When incorporating predictions into an online algorithm, an important consideration is deciding what to predict. At a high level, the predictions should give a parsimonious representation of the problem instance. Specifically, we want the prediction to satisfy three properties:

- The performance of the algorithm should degrade gracefully with the error in predicted quantities.
- The predictions should be robust to inconsequential changes in the problem instance.
- The predictions should be efficiently learnable, that is they should be concise and come from a limited domain.

The first point is critical to good algorithm design. Machine learned approaches are never perfect, and errors are to be expected, and any algorithm that is not robust to errors in the predictions is bound to perform poorly in practice. In their definition of the model, Lykouris and Vassilvitskii [108] further insist that algorithms are *consistent*, that is they recover the optimal solution as the prediction error goes to zero.

The next two properties describe what it means for the learning to be meaningful. For instance, some quantities may be easy to predict, but give no insight into the structure of the problem instance. For instance, predicting the total load on each machine in the optimal solution is *not* a good prediction—one can easily extend any example to make sure that the total load on each machine is identical. We give other examples of poor options for predictions, and further describe the qualities of a meaningful prediction in Section 3.2.3.

In contrast, the last point puts limits on the nature of the predictions, making sure they can be efficiently learned. For instance, one may propose predicting the exact instance that will appear and then execute the offline algorithm on the learned prediction. The field of computational learning theory has developed strong bounds on the number of examples needed to effectively learn a function from a given class. Specifically, it is well known that if the hypothesis class contains N functions, one needs at least $\log N$ examples to learn the best function (this is the weakest lower bound, typically many more examples are necessary) [117]. In the case of restricted assignment, each of the jobs can be matched to 2^m machines, therefore an instance with n jobs arriving, can take on 2^{mn} different values. Thus learning the jobs requires at least $\Omega(mn)$ examples, which is prohibitively expensive even for relatively small m and n. On the other hand, learning a single parameter per machine is a much easier learning task, requiring many fewer examples.

In much of the previous work the choice of the prediction was relatively straightforward. For instance for the classic ski rental problem, Purohit et al. [126] predict the number of skiing days, and then base their decision on the value of the prediction vis-à-vis the cost of buying the skis. In streaming heavy hitters, Hsu et al. [80] predict whether an element is likely to be a heavy hitter, and if so, maintain its count exactly, rather than resorting to a sketch. Finally, for the online caching problem, Lykouris and Vassilvitskii [108] focus on predicting the subsequent arrival time of each element, and then modify the Marking algorithm to take advantage of this new information. In contrast, in online scheduling, the question of what to predict is not obvious.

3.2.3 Predictions for Online Scheduling

The decision of what to predict obviously influences the design of the algorithm using these predictions. However, even without an algorithm in mind, we can eliminate some choices because they fail to satisfy one of the criteria listed above.

For the online scheduling problem, the quantity we predict should intuitively guide us how congested a particular machine is going to be. Consider a restricted version of the problem, where the optimal makespan has value one. Then each instance is equivalent to a bipartite graph between jobs and machines, and the offline problem is to compute a matching between jobs and machines. Given the full instance, what is a good representation that can be used to guide the online algorithm?

One natural approach is to look at the degree of each machine, i.e. the number of jobs that *could* be assigned to it. However, by adding a small number of dummy jobs and machines it is easy to modify each instance so that all machines have identical degree, in which case, this prediction does not carry any additional information. A similar fate befalls predictions of the load of each machine in the optimal solution, the degree of the jobs, and other simple heuristics: for each of these there exists a simple transformation that makes this additional information vacuous.

A more robust approach is to look at the dual problem, and consider learning the dual variables corresponding to machines. This kind of a setting has been successfully used for online bipartite matching—Devanur and Hayes [54] showed how to use duals learned on a random sample of the input to give approximately optimal solutions in an online setting. Critically, however, the choice of the objective plays a large role: while using $1 + \epsilon$ approximate duals gives a $(1 - O(\epsilon))$ solution to the bipartite matching, the same approach does not yield a constant competitive approximation to the makespan. The reason is that in job scheduling, we must match *all* of the jobs to machines and compare the resulting makespans, whereas in online matching, we only try to match as many jobs as possible to empty machines and compare the cardinality of the matching.

Another approach for online matching, advocated by Vee et al. [145], was to formulate the online matching problem as a quadratic program, and then look at the dual variables in that space. In addition to the mismatch in objective described above, we note that the duals in the Vee et al. formulation are extremely sensitive, and approximately correct duals may no longer lead to near optimal solutions on the primal.

3.3 A Robust Online Algorithm via Machine Weights

In this section we identify a quantity that compactly captures the structure of an offline instance of the problem. We give an online algorithm to compute fractional solutions using these quantities, and show that they are robust to errors, if we were to predict them.

The key idea is to assign a *weight* to each machine, and then allocate each job proportionally to the weights of the machines that it can be assigned to. Intuitively, the machine weight is inversely proportional to the contentiousness of the machine, i.e. machines with very high demand have small weights.

Formally, let $w \in \mathbb{R}_+^m$ be a vector of non-negative weights, one per machine. Let $x_{ij}(w)$ denote the fractional assignment of job j on machine i when using weights w, we define the assignment function as:

$$x_{ij}(w) = \frac{w_i}{\sum_{i' \in N(j)} w_{i'}} \tag{3.1}$$

We first need to show that the weights capture enough information for us to reconstruct a good solution. In other words, we need to ensure that for any offline instance there are a set of weights such that the corresponding fractional assignment has a near optimal makespan. We build upon the work of Agrawal et al [5] who showed the existence of such weights for b-matchings. Formally, we say that a set of weights w is c-**good** if for every machine i, $\sum_{j \in N(i)} p_j x_{ij}(w) \leq cT$ for some constant $c \geq 1$. In Section 3.7 we show that good weights exist for arbitrarily small c for the restricted assignment problem.

Given that weights are enough to reconstruct approximately optimal solutions, they will be our target for predictions. Before analyzing what happens when weights are predicted incorrectly, we observe that the allocation given by w is *scale invariant*, i.e. that γw yields the same allocation as w for any $\gamma \in \mathbb{R}$.

Remark 3.3.1. *The fractional assignment produced by w is scale invariant, i.e. for any $\gamma \in \mathbb{R}$, $x_{ij}(w) = x_{ij}(\gamma w)$.*

3.3.1 Constructing Fractional Solutions Online Using Learned Weights

Suppose that $\hat{w}$ is a prediction of good weights w. Due to scale invariance, we can assume that $\hat{w}_i \geq w_i$ for all i. First we consider using $\hat{w}$ directly to compute allocations online using Equation 3.1. To analyze this procedure we define $\mu_i = \hat{w}_i / w_i \geq 1$ to be the relative error with respect to the i'th machine. We define the total error in the prediction to be $\eta = \max_i \mu_i$. Consider the allocation $x_{ij}(\hat{w})$ given by the predictions. Intuitively, this allocation is locally a good approximation to $x_{ij}(w)$, which implies that the makespan of our complete solution is bounded. The following claim makes this intuition precise and implies that the naive algorithm will have a makespan of $O(\eta T)$.

Claim 3.3.2. *For all i and j we have that $x_{ij}(\hat{w}) \leq \eta x_{ij}(w)$.*

Proof. Using equation (3.1) we have:

$$\begin{aligned} x_{ij}(\hat{w}) &= \frac{\hat{w}_i}{\sum_{i' \in N(j)} \hat{w}_{i'}} = \frac{\mu_i w_i}{\sum_{i' \in N(j)} \mu_{i'} w_{i'}} \\ &\leq \eta \left(\frac{w_i}{\sum_{i' \in N(j)} w_{i'}} \right) = \eta x_{ij}(w). \end{aligned}$$

□

Thus we see that the error in our prediction cleanly shows up in the competitive ratio of our algorithm. However we can use standard techniques from online algorithms to improve on this result exponentially. The key idea is that we do not have to continue using the current predictions $\hat{w}$ if we believe the error is large. Rather than use the predictions statically, we update them over time to account for errors that have been detected. The following observation is important in formalizing this idea. If for all i $1/2 \leq \hat{w}_i/w_i \leq 1$, then $x_{ij}(\hat{w}) \leq 2x_{ij}(w)$. This follows by applying the same style of analysis as above. This motivates the design of Algorithm 6.

Algorithm 6 Improved algorithm for computing fractional assignments online

```
Let $\hat{w}$ be predictions of $w$
Initialize $L_i \leftarrow 0$ for each machine $i$          ▷ $L_i$ = fractional load of machine $i$
for each job $j$ do
    For all $i \in N(j)$, $L_i \leftarrow L_i + p_j x_{ij}(\hat{w})$          ▷ Compute fractional assignment
    for $i = 1, \dots, m$ do
        if $L_i > 2T$ then
            $\hat{w}_i \leftarrow \hat{w}_i/2$, $L_i \leftarrow 0$          ▷ Update $\hat{w}$, start new phase
        end if
    end for
end for
```

Algorithm 6 keeps track of the load of each machine in phases. At the start of a machine's phase its load L_i is initialized to 0. As each job j arrives we update L_i by adding $x_{ij}(\hat{w})$. At this point if $L_i \leq 2T$ we do nothing, as we have no reason to believe that $\mu_i = \hat{w}_i/w_i$ is very large. However, if $L_i > 2T$, then by our observation we know that μ_i is large, so we update $\hat{w}_i$ by dividing it by 2 and start a new phase by resetting L_i to 0. Once the condition in our observation is satisfied, we know that this will be the last phase for all machines. So the question becomes, how many phases will each machine go through until the condition is satisfied. Let k_i be the number of phases that machine i starts. The condition is satisfied for machine i when $1/2 \leq \frac{\hat{w}}{2^{k_i} w_i} \leq 1$

which implies that $k_i \leq \lceil \log_2(\hat{w}_i/w_i) \rceil + 1$. All machines satisfy the condition after $k = \max_i k_i = \max_i \log_2(\hat{w}_i/w_i) = \max_i \log_2(\mu_i)$ phases.

How much does this algorithm lose in terms of the makespan? Each machine phase incurs a factor of 2 loss in the makespan, and each machine has at most $k = \max_i \log_2(\mu_i)$ phases. Thus we see that the resulting makespan is $O(kT) = O(\max_i \log_2(\mu_i)T)$. Since $\mu_i \geq 1$, we have that $\max_i \log_2(\mu_i) = \log_2(\max_i \mu_i) = \log_2(\eta)$. Thus the competitive ratio is $O(\log_2(\eta))$, an exponential improvement over naively using the predictions. We state the above results in Theorem 3.3.3.

Theorem 3.3.3. *Let $\hat{w}$ be predictions of a set of good machine weights w and let $\eta = \max_i \hat{w}_i/w_i$ be the maximum error in our predictions. Then Algorithm 6 is an $O(\min\{\log \eta, \log m\})$-competitive algorithm for minimizing the fractional makespan online.*

In order to get the stated competitive ratio of $O(\min\{\log \eta, \log m\})$, we run Algorithm 6 normally until assigning some job causes our algorithm to have makespan $> 2T \log m$. In this case, the predictions are not helpful and we switch to a $O(\log m)$-competitive algorithm from the literature, such as the ones described in [21] or [18].

3.4 Rounding Algorithm Overview

Before delving into the technical details, we first describe an overview of the rounding algorithm. Recall that jobs arrive online and when job j arrives the algorithm learns x_{ij} for all machines i. We assume that $\sum_{i \in [m]} x_{ij} = 1$ and the goal is to be competitive with the final fractional makespan. $T := \max_{i \in [m]} \sum_{j \in [n]} p_{ij} x_{ij}$. To make the exposition simpler, we discuss the case of restricted assignment with unit sized jobs and the algorithm knows the exact fractional makespan T a priori. In this case, each job has size 1 or ∞ on each machine.

First observe that if $T \geq \Omega(\log m)$ then the rounding is easy. Each job j independently performs randomized rounding, selecting a machine i with probability x_{ij}. Because the contribution of each job is much smaller than the total makespan, standard concentration bounds ensure that the makespan is bounded by $O(T)$ with high probability. The challenging case is when T is small compared to $\log m$. In the proof, we denote this the "large" job case.

We further break up the analysis into two cases. Let $B_j = \{i \mid x_{ij} \geq \frac{1}{\log^2 m}\}$ and $S_j = \{i \mid x_{ij} < \frac{1}{\log^2 m}$ and $x_{ij} > 0\}$. The set B_j contains the machines where x_{ij} is big and S_j are the machines in the support where x_{ij} is small. Let $\mathcal{B}$ be the set of jobs j where $\sum_{i \in B_j} x_{ij} \geq \frac{1}{2}$. These are jobs mostly assigned using large x_{ij} values. Let $\mathcal{S}$ be the remaining jobs. These are jobs assigned using mostly small x_{ij} values.

Jobs in $\mathcal{B}$. We begin by simplifying the instance. We show that at the cost of losing a factor of $O(\log \log m)$ in the total cost, we can transform the instance to one with $x_{ij} \in \{0, \frac{1}{\lambda}\}$ for a single value $\frac{1}{\log^2 m} \leq \frac{1}{\lambda} \leq 1$. This further implies that each job j has $\Theta(\lambda)$ machines in the support of x_{ij}. Let $N(j)$ be this set of machines. Notice that each machine can have at most $\tilde{O}(T\lambda) = O(\operatorname{poly}\log m)$ jobs that can be assigned to it. This case is hard because the fractional solution is revealing very little information. Indeed, each job is uniformly split across a neighborhood of size λ.

To reason about the rounding in this case, consider the bipartite graph G corresponding to the problem instance. Nodes representing jobs are on one side, and machines are on the other, with an edge between a job and machine if the (job, machine) pair is in the support. By construction the maximum degree in this graph, $\Delta = O(\operatorname{poly}(\log m))$. Now to allocate jobs we use the following algorithm: when job j arrives, it selects machines independently at random from $N(j)$, selecting machine i with probability $\Theta(\log\log(m) \cdot x_{ij})$[2]. The job can assign itself to any machine chosen so long as the machine has been assigned at most $O(T \log\log m)$ jobs so far. If the job does not select a machine or the machines are overloaded then the job "fails" and enters a set F of failed jobs.

Let G_F be the induced subgraph consisting only of the failed jobs and all of the machines. The key to the proof is showing that with high probability *every* connected component in G_F is small. In particular, each connected component has fewer than $\operatorname{poly}\log m$ nodes. Intuitively, this is because the graph G has maximum degree at most $\operatorname{poly}\log m$ and therefore the graph is broken into small pieces. This is reminiscent of the shattering idea in the parallel graph algorithms community [33, 67].

If this is the case, and the components are small then the problem becomes easy. Each failed job assigns itself greedily to the least loaded machine. It is known [21] that the greedy deterministic algorithm is a $O(\log m')$-approximation for any input with m' machines. We can think of each component as an individual instance, resulting in $m' \leq \operatorname{poly}\log m$ and these jobs contribute at most a $O(T \log\log m)$ amount to the makespan.

Finally, we argue that the components are indeed small. Notice that if there is a connected component of size $\operatorname{poly}\log m$ then there should exist a path in the graph of length at least $\frac{\operatorname{poly}\log m}{\lambda}$ because the maximum degree is λ. Thus, it suffices to show that no such path survives. The proof begins by establishing that each job fails with probability at most $\frac{1}{\log^c m}$ for some constant c by simple concentration bounds. This means that every edge in G remains in G_F with probability at most $\frac{1}{\log^c m}$.

For sake of intuition assume that each edge were to be removed *independently* with this probability (this is not true and we will remove this assumption shortly). The probability a fixed path of length ℓ survives is at most $(\frac{1}{\log^c m})^\ell = \frac{1}{\log^{c\ell} m}$. Hence we can union bound over all possible paths to show that no long paths survive. More precisely, assume that the path starts at a machine node in G_F and there are m starting positions. Recall that the maximum degree is Δ, thus the total number of paths of length ℓ is bounded by $m\Delta^\ell$. Ensuring that $\Delta \leq \log^3 m$ and choosing $c \geq 4$ and $\ell \geq \log m$ ensures no path exists with good probability. This implies there is no large connected component, completing the analysis of jobs in $\mathcal{B}$.

The only issue that remains is the assumption on the independence of the edges. The proof establishes that edges or nodes sufficiently far apart survive to be in G_F independently.

[2]Note that machines are chosen independently and this independence is crucial for the proof. A job may select more than one machine or no machines. It easily follows that the probability a job fails to chose a machine is bounded by $\exp(\Theta(-\log\log m))$

By carefully counting 'special' sets of edges that survive independently because they are well separated, but still reachable in a few hops, allows us to effectively use the above argument.

Jobs in $\mathcal{S}$. When we rely on machines with small assignment, we will run randomized rounding in phases. In phase k, each job j that makes it to the phase selects a machine with probability x_{ij}. If the chosen machine's makespan is smaller than $O(T)$ from jobs assigned during phase i then the job goes to the machine. If not, then the job goes to phase $k+1$.

Define the fractional makespan of phase k to be the maximum fractional makespan on the machines only counting jobs that survive to phase k. Using concentration bounds, we can show that the fractional makespan decreases rapidly with each phase. Intuitively, this is because most jobs have a good probability of being successfully assigned. After $O(\log\log m)$ number of phases, the fractional makespan will drop below $O(\frac{1}{\log^2 m})$. This is the last phase. At this point, if a job still survives then the job chooses $\log m$ machines in $N(j)$ uniformly at random. Then the job goes to a machine that no other job from this phase selected. Because the fractional makespan is so small, concentration bounds will imply that with high probability one of the machines that each job picks with be chosen only by that job. Thus, the overall the makepsan will be $O((\log\log m)T)$ from rounding jobs in $\mathcal{S}$ with high probability.

3.5 Online Rounding Algorithm & Analysis

In this section we give a formal analysis of the online rounding algorithm. For this section we assume the more general unrelated machine problem. Recall the setup of the problem. Jobs arrive over time online. When each job j arrives, the value of the fractional assignment, x_{ij}, and the job size p_{ij} is revealed for all machines i. That is, at each time t we know the fractional assignment x_{ij} for all jobs j that have arrived up to time t and have no information about the future jobs. We assume that $\sum_i x_{ij} = 1$. That is, all jobs are fully fractionally assigned.

The goal is to assign jobs online to machines integrally using the fractional values as a guide so that the final makespan is as close as possible to the makespan of the fractional schedule. Since the integrality gap can be bad for the underlying linear relaxation[3], we define the quantity $T := \max\{\max_{i\in[m]} \sum_{j\in[n]} p_{ij}x_{ij}, p^*\}$, where $p^* = \max\{p_{ij} \mid x_{ij} > 0\}$. Note that this assumption enforces $p_{ij} \le T$ whenever $x_{ij} > 0$. It is known that the integrality gap is large if this condition is not met [149]. Since we apply the result of this section to the case of restricted assignment, we note that the definition of T above reduces to $T = \max\{\max_{i\in[m]} \sum_{j\in N(i)} p_j x_{ij}, \max_j p_j\}$ for this case.

An interesting challenge in our setting is that the assignment needs to be online so the algorithm only has partial knowledge of the instance. We assume no structural properties

[3]If there is 1 unit size job with $x_{ij} = 1/m$ for all $i \in [m]$ then any assignment has makespan 1, a factor of m larger than the fractional makespan.

on the fractional solution. In particular, we do *not* assume that the fractional assignment corresponds to a vertex of the linear program for makespan on unrelated machines, a key property used in offline rounding procedures [105, 136].

Now we present an online randomized algorithm for rounding fractional assignments which achieves a competitive ratio of $O(\text{poly}(\log\log m))$ with high probability. Throughout the analysis, we will assume that T is known. Later we discuss how we can remove the assumption on the knowledge of T using a standard doubling analysis. We state our result formally as the following theorem.

Theorem 3.5.1. *Let x be a fractional assignment of unrelated machines that is received online and let T be the fractional makespan of x, i.e. $T := \max_i \sum_{j\in N(i)} p_{ij}x_{ij}$. Further, $x_{ij} = 0$ if $p_{ij} > T$. There exists a randomized online algorithm that rounds a fractional assignment to an integer assignment such that the resulting makespan is at most $O((\log\log m)^3 T)$ with high probability.*

3.5.1 Instance Transformation

The first step in our analysis is to convert the instance into a number of simpler instances as we receive it online. Depending on the properties of the job, it will be sent to a procedure for that particular job type.

We redefine the neighborhood for the unrelated case as $N(j) := \{i \mid x_{ij} > 0\}$ be the set of machines in the support for job j. We will refer to this as the **neighborhood** of job j. Recall that job sizes are bounded in the following way: $p_{ij} \le T$ for all $i \in N(j)$. Now the first breakdown we make is to separate jobs into a notion of small and large jobs. For a job j let $S_j = \{i \in N(j) \mid p_{ij} \le T/\log m\}$. We say that a job is small if $\sum_{i\in S_j} x_{ij} \ge 1/2$, and otherwise it is large. Intuitively, a job is small if most of its fractional weight is on machines with small p_{ij} as compared to T. Note that this separation can easily be done online because it only depends on p_{ij} and x_{ij} for a job j.

The interesting case is the large jobs, which we discuss next. The small jobs can be assigned by using randomized rounding as we show in Section 3.5.2. Because the jobs are small, Chernoff bounds ensure no machine is overloaded with high probability.

Transforming Large Jobs

We first consider how to round the large jobs. For this, we further break the jobs into cases. For each job j let $B_j = N(j) \setminus S_j$ be the set of machines i in $N(j)$ where $p_{ij} > T/\log m$. Let $\mathcal{B}$ be the set of large jobs. For each large job j we have $\sum_{i\in B_j} x_{ij} \ge 1/2$. We now preprocess the large jobs online creating a new fractional solutions x' where the following properties hold.

Lemma 3.5.2. *At a loss of increasing the makespan by a $O(\log\log m)$ factor, the fractional solution x can be converted to a fractional solution x' where the following properties hold:*

- $x'_{ij} \ge 0$ *and* $\sum_{i\in N(j)} x'_{ij} = 1$
- $x'_{ij} \le 2\log\log(m)x_{ij}$

- *If $x'_{ij} > 0$ then $p_{ij} = 2^k T/\log m$ for some fixed $k \in [\log\log m]$*

This modification can be done for each job individually in an online manner.

This preprocessing step will allow us to assume that the size of the job is the same on all machines in the support of x' for the job.

Proof. Consider the intervals $I_k = [2^{k-1}\frac{T}{\log m}, 2^k \frac{T}{\log(m)}]$ for $k \in [\log\log m]$. We have that $[T\log m, T] = \bigcup_{k=1}^{\log\log m} I_k$. Let $B_{j,k} = \{i \in B_j \mid p_{ij} \in I_k\}$. Since all large jobs have most of their fractional weight on machines with $p_{ij} \in [T/\log m, T]$, by averaging there is a $k \in [\log\log m]$ such that $\sum_{i\in B_{j,k}} x_{ij} \geq \frac{1}{2\log\log m}$. That is, a large fraction, at least $\frac{1}{2\log\log m}$, of a job's fractional assignment is to machines where the sizes are within a factor 2 of each other. Set $x'_{ij} = 0$ for $i \notin B_{j,k}$ and $x'_{ij} = x_{ij}/\sum_{i'\in B_{j,k}} x_{i'j}$ for $i \in B_{j,k}$. It is simple to verify the above properties for this transformation.

Since for all i such that $x'_{ij} > 0$ we have that $p_{ij} \leq 2^k T/\log m$, we can think of the job as having a single size $p'_j = 2^k T/\log m$ on its neighborhood of machines for some $k \in [\log\log m]$ by rounding the size up by at most a factor two. $\square$

Thus we have reduced the more general unrelated machines instance to an instance of restricted assignment. In the new restricted assignment instance, a job has a fixed size, but can only be assigned to a subset of machines.

Let C_k be the set of large jobs in the k'th class that now have size $2^k T/\log m$. We say that $j \in C_k$ is of **class** k. In the remainder of this section we show how to round the jobs in the k'th class with small loss in the makespan. Since there are $O(\log\log m)$ such classes and we increased each fractional value by at most an $O(\log\log m)$ factor, we lose an extra factor of $O((\log\log m)^2)$ overall. For simplicity we assume throughout the rest of the analysis that the solution x has the properties stated in the claim.

3.5.2 Rounding A Single Class of Large Jobs

We now focus on a single class C_k of large jobs. All jobs in this class have the same size $p'_j = 2^k T/\log(m)$, but a job specific neighborhood $N(j)$ of feasible machines. We break these down into two more cases.

For a job j let $S'_j = \{i \in N(j) \mid x_{ij} \leq \frac{1}{\log^2 m}\}$. We say that a job j's fractional assignment has small support if $\sum_{i\in S'_j} x_{ij} \geq 1/2$, and otherwise it has large support. (This inference can be easily done online). We start by analyzing the jobs with large support.

Jobs with Large Support

For jobs with large support, we further preprocess the instance to give it more structural properties. In particular, we will show that by increasing the makespan by a $\log\log m$ factor we can assume that for a fixed job j the values of x_{ij} are either 0 or a single positive value.

Lemma 3.5.3. *Fix a class C_k of large jobs. The fractional solution can be modified by increasing the makespan by a factor $O(\log\log m)$ to ensure the following property holds. For each job $j \in C_k$, for all i either $x_{i,j} = 0$ or $x_{i,j} = \frac{2^\ell}{\log^2 m}$ for some fixed $\ell \in [\log\log m]$. This modification can be done for each job individually in an online manner.*

Proof. For a job with large support we have that most of its fractional assignment is on machines i with $\frac{1}{\log^2 m} \leq x_{ij} \leq 1$. Fix a job j. Consider grouping machines by their fractional values in powers of 2. A machine is in **group** $\ell \in [2\log\log m]$ for job j if $x_{ij} \in [\frac{2^\ell}{\log^2 m}, \frac{2^{\ell+1}}{\log^2 m}]$. Let $\mathcal{G}_{j,\ell}$ contain all such machines.

By an averaging argument, there is a group $\mathcal{G}_{j,\ell}$ of machines where the job j has at least a $1/2\log\log m$ of its fractional assignment. That is, there is an $\ell \in [2\log\log m]$ where $\sum_{i\in\mathcal{G}_{j,\ell}} x_{ij} \geq \frac{1}{4\log\log m}$. Let λ_j be the number of machines in this group. Since all the fractional assignments in this group are off by at most a factor of 2 from each other, we might as well consider them to be the same at the cost of a factor of 2. We set $x'_{ij} = 1/\lambda_j$ to be the new fractional assignment for machines in this group and $x'_{ij} = 0$ for machines outside of this group. By construction we have that $\frac{\log^2 m}{2^\ell} \leq \lambda_j \leq \frac{\log^2 m}{2^{\ell-1}}$ for some ℓ. Let D_ℓ be the set of large support jobs with λ_j in the interval $[\frac{\log^2 m}{2^\ell}, \frac{\log^2 m}{2^{\ell-1}}]$. We will refer to a set D_ℓ of jobs as a **group** for some fixed ℓ. Since there are $O(\log\log m)$ such groups of large support jobs, it suffices to consider only a single such group at the cost of increasing the makespan by a $O(\log\log(m))$ factor. □

Rounding a Single Group of Large Support Jobs

This section gives the algorithm for the case where jobs have large support. Fix a class D_ℓ of large support jobs. All of these jobs have a neighborhood of size at most λ for some $\lambda \leq \log^2 m$. Our aim is to show that a single iteration of randomized rounding followed by a deterministic greedy assignments suffices to assign these jobs in a good way.

The algorithm we use to round these jobs is as follows. Each job j chooses a random machine in its neighborhood $N(j)$, then checks the machine it chose. If the load incurred by other jobs in class D_ℓ on this machine is greater than $101\log\log mT$, then the job rejects this assignment. In this case the job is added to the set F_ℓ of failed jobs for class ℓ. The jobs in F_ℓ are assigned using a deterministic greedy algorithm.

The greedy algorithm works as follows. For a machine $i \in N(j)$ let its ℓ-load be the number of jobs in F_ℓ that have already been assigned to it. A job $j \in F_\ell$ chooses to be assigned to the a machine with the minimum ℓ-load. This can easily be done online.

By definition, the jobs assigned using randomized rounding contribute $O((\log\log m)T)$ to the makespan. Thus it suffices to bound the contribution of the jobs assigned using greedy.

Let G_ℓ be the bipartite graph consisting of nodes for each job in F_ℓ that rejected their random assignment. The set of machines are on the other side. A job $i \in F_\ell$ is connected to a machine j with an edge if and only if $x_{i,j} \geq 0$. The proof will show that every connected component of G_ℓ has size $O(\text{poly}(\log m))$ with high probability. It then follows that the jobs assigned by greedy contribute $O(\log\log m)T$ to the makespan. This is because the deterministic greedy algorithm [21] is known to achieve a $O(\log \hat{m})$ approximation for any instance of restricted assignment on $\hat{m}$ machines and each connected component is an instance of size $O(\text{poly}(\log m))$ with high probability. Thus, these are "instances" of size $O(\text{poly}(\log m))$ and the makespan of the greedy algorithm as compare to optimal can be at most a $O(\log\log m)T$ factor larger.

In order to show that G_ℓ has small connected components we apply a technique similar to shattering in the distributed computing literature. We will define a special substructure and show that if a connected component of G_ℓ is large, then one of these substructures exist. We will then show that the probability of one of these substructures existing is small; after carefully counting the number of possible substructures and applying a union bound we can conclude that every connected component of G_ℓ is small with high probability.

We start by defining the substructure.

Definition 3.5.4. *Two jobs j and j' are machine disjoint if $N(j) \cap N(j') = \emptyset$.*

Definition 3.5.5. *A sequence $j_1, j_2, \ldots, j_\beta$ of jobs is* ***special*** *if all the jobs are pairwise machine disjoint and for each k, j_k is within 4 hops of at least one of $j_1, \ldots, j_{k-1}$ in G.*

Later on, machine disjointness will allow us to show that certain events are statistically independent. Note that by definition, all of the jobs in a special sequence must belong to the same connected component. Equipped with these definitions we prove the following lemma:

Lemma 3.5.6. *Let C be a connected component of G_ℓ of size at least $\log^c m$ with $c > 7$, then there is a special job sequence of size $\beta = \log m$.*

Proof. We prove the lemma by induction on the size of the sequence. Every large connected component has at least one job, so the base case is trivial. Now for the induction step. Let C be a connected component with size at least $\log^c(m)$. Suppose there is a special job sequence of $\beta - 1$ jobs. We combine these jobs into a single node and start a breadth first search in C. Since the underlying graph is bipartite, the first level of this search consists of machines, the second jobs, and the third machines. If there is any job in the fourth level of this search, then it must be machine disjoint from the first $\beta - 1$. Suppose that there is no such job. Then 3 levels of this search suffices to explore all nodes of C. Thus since the maximum degree of a job or machine is bounded by λ, the size of C is at most:

$$|C| \leq (\beta - 1)\lambda^3 \leq (\beta - 1)\log^6 m.$$

This leads to a contradiction when $\beta = \log m$ and $c > 7$. Thus such a job in the fourth level exists, yielding a special job sequence of size $\beta = O(\log m)$. □

The proof above gave a way to construct the sequence of jobs given the graph. This also gives us a way to upper bound the number of such special sequences.

Lemma 3.5.7. *Let C be a connected component of G_ℓ. There are at most $m(\beta\lambda)^{4\beta}$ special job sequences for $\beta = \log m$. So for $\lambda \leq \log^2 n$, this is upper bounded by $m(\log^{12} m)^{\log m} \leq m^{1+12\log\log m}$.*

Proof. Let us count special job sequences by following the construction given in the proof of Lemma 3.5.6. There are at most m jobs we can start with to construct a special sequence. At the i'th step of the construction there are at most $(i\lambda)^4$ possible jobs we can choose to append onto the sequence by traversing out 4 hops in the graph from the currently chosen jobs. Since $i \leq \beta$, this is at most $(\beta\lambda)^4$. Thus there are at most $m(\beta\lambda)^{4\beta}$ such special job sequences. □

The previous lemma bounds the number of possible special job sequences. Using this, we can bound the probability that all jobs in a fixed special job sequence fail to be assigned by randomized rounding, and then union bound over all possible special job sequences.

Lemma 3.5.8. *Fix a special job sequence σ of length $\beta = \log m$. The probability that all jobs in σ fail to be assigned by randomized rounding is at most $m^{-100 \log\log m}$.*

Proof. Recall that a job is failed to be assigned by randomized rounding if it chose a machine with load $> (100 \log\log m + 1)T$. Let L_i be the load of jobs that sample machine i in randomized rounding and let X_{ij} be the random variable indicating whether or not job j sampled machine i in randomized rounding. Then we have $L_i = \sum_{j \in N(i)} p'_j X_{ij}$ and in expectation:

$$\mathbb{E}[L_i] \leq \sum_{j \in N(i)} p'_j \frac{1}{\lambda} \leq T$$

Applying Theorem 3.2.3, we have that the probability that a machine becomes overloaded is:

$$\begin{aligned} &\Pr[L_i > T(1 + 100 \log\log m)] \\ &\qquad \leq \exp\left(-\frac{(100 \log\log m)^2}{2 + \log\log m}\right) \\ &\qquad \leq \exp(-100 \log\log m) \end{aligned}$$

Now let σ be a special job sequence. For $j \in \sigma$, we have that the probability that j fails to be assigned by randomized rounding is at most the probability that any machine becomes overloaded. Since $N(j) \cap N(j') = \emptyset$ for all $j, j' \in \sigma$, we have that the jobs in σ fail to be assigned by randomized rounding independently. Thus the probability that all jobs in σ fail to be assigned is at most

$$\begin{aligned} \exp(-100 \log\log m)^\beta &= \exp(-100\beta \log\log m) \\ &= m^{-100 \log\log m}, \end{aligned}$$

proving the lemma. □

We are now ready to show that every connected component in G_ℓ is small with high probability.

Lemma 3.5.9. *With high probability, every connected component of G_ℓ has size $O(\log^c m)$.*

Proof. By Lemma 3.5.6, it suffices to show that no special sequence of jobs of length $\beta = \log m$ exists in G_ℓ with high probability. For a fixed special sequence of jobs, Lemma 3.5.8 states the probability that it is in G_ℓ is at most $m^{-100 \log\log m}$. Now taking a union bound over all special sequences, the probability that there exists a special sequence of jobs in G_ℓ of length $\beta = \log m$ is at most

$$\begin{aligned}\left(m^{-100\log\log m}\right)\left(m(\log^{12} m)^{\log m}\right)& \\ &= \left(m^{-100\log\log m}\right)\left(m^{1+12\log\log m}\right)\\ &\le m^{-5},\end{aligned}$$

where we use the bound on the number of special job sequences from Lemma 3.5.7. Thus no special sequence of length $\log m$ exists in G_ℓ with high probability. $\square$

Lemma 3.5.10. *Fix a class ℓ of large support jobs. We can round the jobs in this class with makespan at most $O(\log\log m)T$ with high probability.*

Proof. Each job in this class is assigned by randomized rounding or by a separate greedy assignment. The jobs assigned by randomized rounding contribute $O(\log\log m)T$ to the makespan by definition of our algorithm. Looking at the instance after removing all jobs assigned by randomized rounding, by Lemma 3.5.9 every connected component in the underlying graph of this instance has size at most $O(\log^c m)$ for some constant c with high probability. Thus running Greedy on this remaining instance contributes an extra $O(\log\log m)T$ to the makespan. In aggregate, the contribution to the makespan from this class of large support jobs is $O(\log\log m)T$. $\square$

Rounding Jobs with Small Support

In this section we consider the case where for jobs j that have small support. Recall that this means that $\sum_{i\in S'_j} x_{ij} \ge 1/2$ where S'_j is the set of machines i where $x_{ij} \le \frac{1}{\log^2 m}$. In this case, we set $x_{ij} = 0$ for $i \notin S'_j$. Then we re-normalize the remaining fractional assignment to ensure $\sum_i x_{ij} = 1$ for all j, increasing each assignment by at most a factor of 2. Note that since we are rounding jobs of a single class, we may assume all jobs are unit sized and the makespan is T is bounded by $O(\log m)$ by rescaling.

The algorithm that we use to handle this type of jobs, Iterated Randomized Rounding, works in several phases. Each phase k maintains a fractional load T_k and integer load $L(i,k)$ for each machine. The load $L(i,k)$ counts the total size of jobs assigned using this procedure in phase k. In each phase we attempt to randomly assign a job to several machines, however this fails if $L(i,k)$ is too large for the sampled machines. In the case of failure, the job goes on to the next phase. Our analysis will show that after $O(\log\log m)$ phases only few jobs will be left and we will handle them separately.

Interestingly the procedure can be done for each job individually, where $L(i,k)$ is the load assigned to the machine so far among jobs in phase k. Thus the procedure can be done online.

In the first phase of Iterated Randomized Rounding, the fractional load of each machine is at most T. The intuition behind this algorithm is that as the algorithm goes to higher

Algorithm 7 Iterated Randomized Rounding

```
1: for each job $j$ do
2:     for each phase $k = 0, 1, 2, \dots$ do
3:         For each $i \in N(j)$ assign $j$ to $i$ independently with probability $x_{ij}$
4:         If $L(i,k) > 10T$ for all sampled machines, then $j$ goes to the next phase
5:         Otherwise assign $j$ to $i$ such that $L(i,k) < 10T$ and increase the load of all
       sampled machines $i$ with $L(i,k) < 10T$
6:     end for
7: end for
```

and higher phases, then this fractional load should decrease quickly. Let T_k be the bound on the fractional load in phase k of Iterated Random Rounding. We will to show that $T_{k+1} \leq \rho T_k$ for some constant $\rho \in (0,1)$ with high probability. We can continue running Iterated Random Rounding while this bound is relatively large. When we reach a phase where the fractional load becomes too small, any job that is still unassigned becomes a "leftover" job and we assign it using a different technique. The number of phases of Iterated Randomized Rounding we need will be $O(\log\log m)$, implying that the contribution to the makespan of jobs with small support will be $O(\log\log m)T$ plus the contribution of "leftover" jobs. We start the analysis with the following lemma.

Lemma 3.5.11. *Let T_k be an upper bound on the fractional load of all jobs that make it to phase k in Iterated Randomized Rounding, with $T_0 = T$. For all $k = 0, 1, 2, \dots$ we have that $T_{k+1} \leq \rho T_k$ for some constant $\rho \in (0,1)$ with high probability.*

Proof. Consider a phase k and let T_k be as in the lemma statement. We need to upper bound the fractional load of jobs that fail to be assigned in phase k, and hence contribute to the fractional load in phase $k+1$. Let $L(i,k)$ be the load of machine i in phase k and let $N(i,k)$ be the set of jobs that can be assigned to machine i in phase k. We have that

$$\mathbb{E}[L(i,k)] = \sum_{j \in N(i,k)} p_j x_{ij} \leq T_k$$

The probability that any job fails to be assigned in phase k is at most the probability that machine i has load more than $10T$ in phase k. By Markov's inequality this is at most

$$\Pr[L(i,k) > 10T] \leq \frac{\mathbb{E}[L(i,k)]}{10T} \leq \frac{T_k}{10T} \leq \frac{1}{10}$$

since $T_k \leq T$ for all k. Now fix a machine i^*. We are interested in how much fractional load i^* may potentially contribute to the next phase. Let $Z(k, i^*, i) = \sum_{j \in N(i,k)} x_{i^*j} I(j \text{ picks } i)$. Intuitively, if j picks machine i and j ends up going to phase $k+1$, then j will contribute x_{i^*j} to i^*'s fractional load. First we bound $Z(k, i^*, i)$ with high probability.

Claim 3.5.12. *For each i, $Z(k, i^*, i) \leq \left(\frac{d+1}{\log^2 m}\right) T_k$ with high probability for some constant $d > 0$ when $T_k = \Omega(\frac{1}{\log m})$.*

Proof. In expectation we have

$$\begin{aligned}\mathbb{E}[Z(k,i^*,i)] &= \sum_{j\in N(i,k)} x_{i^*j}\mathbb{E}[I(j \text{ picks } i)] \\ &= \sum_{j\in N(i,k)} x_{i^*j}x_{ij} \\ &\leq \frac{2}{\log^2 m}\sum_{j\in N(i,k)} x_{ij} \leq \frac{2T_k}{\log^2 m}\end{aligned}$$

Since $Z(k,i^*,i)$ is a sum of independent random variables in the form needed for Theorem 3.2.3, we apply this theorem with $a \leq \frac{2}{\log^2 m}$ and $v \leq \frac{2}{\log^2 m}\mathbb{E}[Z(k,i^*,i)]$. Thus taking $\lambda = dT_k/\log^2 m$, we have

$$\begin{aligned}\Pr\left[Z(k,i^*,i) > \frac{2T_k}{\log^2 m} + \lambda\right] &\leq \exp\left(\frac{-\lambda^2}{\frac{4\mathbb{E}[Z(k,i^*,i)]}{\log^2 m} + \frac{4\lambda}{3\log^2 m}}\right) \\ &\leq \exp\left(\frac{-d^2T_k^2}{\frac{4T_k}{\log^2 m} + \frac{4dT_k}{3\log^2 m}}\right) \\ &= \exp\left(-\left(\frac{d^2}{4+4d/3}\right)T_k\log^2 m\right) \\ &= \exp(-c'\log m) = m^{-c},\end{aligned}$$

for some constant c' depending on d. Note from the second to last line to the last line we used the fact that $T_k = \Omega(1/\log m)$. Now choosing c', d large enough and taking a union bound over all machines we see that $Z(k,i^*,i) \leq \frac{d+1}{\log^2 m}T_k$ for all i with probability at least $1 - 1/m^{c'-1}$. ◻

To bound the fractional load in the next phase define $Z(k,i^*) = \sum_i Z_{k,i^*,i}\mathbb{I}(i \text{ overloaded in phase } k)$. Note that this is a bound on the fractional load that survives phase k and hence goes to phase $k+1$. Thus any bound on this random variable that holds for all i^* yields a bound on T_{k+1}.

Claim 3.5.13. *For each i^*, $Z(k,i^*) \leq \rho T_k$ for some $\rho \in (0,1)$ with high probability when $T_k = \Omega(1/\log m)$.*

Proof. For each i^*, in expectation we have

$$\begin{aligned}\mathbb{E}[Z(k,i^*)] &= \sum_i \mathbb{E}[Z(k,i^*,i)]\Pr[i \text{ overloaded in phase } k] \\ &\leq \frac{1}{10}\sum_i\sum_{j\in N(i,k)} x_{i^*j}x_{ij} \\ &= \frac{1}{10}\sum_{j\in N(i,k)} x_{i^*j}\sum_i x_{ij} \\ &= \frac{1}{10}\sum_{j\in N(i,k)} x_{i^*j} \leq \frac{1}{10}T_k.\end{aligned}$$

By Claim 3.5.12, we have that $Z(k, i^*, i) \leq \frac{d+1}{\log^2 m} T_k$ for all i with high probability. Now since $Z(k, i^*)$ is defined as a sum of independent random variables[4], we can again apply Theorem 3.2.3 to this random variable with $a \leq \frac{d+1}{\log^2 m} T_k$ and $v \leq \frac{d+1}{\log^2 m} T_k \mathbb{E}[Z(k, i^*)]$. Taking $\lambda = qT_k$ we have

$$\begin{aligned}
\Pr\left[Z(k, i^*) > \frac{1}{10}T + \lambda\right] &\leq \exp\left(\frac{-\lambda^2}{\frac{2(d+1)T_k\mathbb{E}[Z(k,i^*)]}{\log^2 m} + \frac{2(d+1)T_k\lambda}{3\log^2 m}}\right) \\
&= \exp\left(\frac{-q^2T_k^2}{\frac{2(d+1)T_k^2}{10\log^2 m} + \frac{2(d+1)qT_k^2}{3\log^2 m}}\right) \\
&= \exp\left(-\left(\frac{q^2}{(d+1)/5 + 2q(d+1)/3}\right)\log^2(m)\right)
\end{aligned}$$

Now we choose q such that $\rho = \frac{1}{10} + q \in (0, 1)$ and the above expression becomes sufficiently small, i.e. $\leq m^{-c}$ for some constant $c > 1$. Then taking a union bound over all machines i^*, we have that $Z(k, i^*) \leq (\frac{1}{10} + q)T_k = \rho T_k$ for all i^* with high probability. □

By Claim 3.5.13, we can take $T_{k+1} = \max_{i^*} Z(k, i^*)$, which is at most ρT_k with high probability, proving the lemma. □

Now we have a sequence of bounds $T_0, T_1, \ldots, T_k, \ldots$ that hold with high probability. Note that in the proof of Claim 3.5.12, we required that $T_k = \Omega(1/\log m)$. Thus it only makes sense to consider this sequence while this bound is true. We assume that $T_0 = T = \Omega(1)$ and that $T = O(\log m)$. Thus there are $O(\log\log m)$ phases before T_k becomes $O(1/\log m)$. Jobs that make it this far without being assigned become "leftover" jobs and we assign them using a different technique.

Rounding the "Leftover" Large Jobs with Small Support

The "leftover" jobs are small support jobs that survived too many phases of random assignments. The setup of this case is the following. Each job has $x_{i,j} \leq \frac{1}{\log^2 m}$. Let $T(i) = \sum_j x_{i,j}$ be the fractional load of machine i. It is the case that $T(i) \leq \frac{1}{64\log m}$ for each machine $i \in m$.

Consider the following algorithm. Each job j independently samples a set $M(j)$ of machines from $N(j)$ where machine $i \in N(j)$ is in the set with probability $32\log m \cdot x_{i,j}$. Each job j is assigned to the machine which has the smallest load in this phase. Now we show that with high probability that for each job j it is always the case that $M(j)$ contains a machine i such that $i \notin M(j')$ for all other jobs j. Thus, with high probability each machine is assigned at most one job.

Lemma 3.5.14. *With probability at least $1 - \frac{1}{m}$ it is the case that for all jobs j the set $M(j)$ contains a machine i not in $M(j')$ for all $j' \neq j$.*

[4]Machines in the same phase become overloaded independently of one another because the machines are selected independently by each job and a job can increase the load of multiple machines in the same phase (even if it is assigned to a single one).

Proof. Fix any job j. First we show that $|M(j)| \geq 5 \log m$ with probability at least $1 - \frac{1}{m^4}$. Indeed, let X_i be 1 if job j samples machine i and 0 otherwise. By definition of the algorithm $\mathbb{E}[X_i] = 32 \log m \cdot x_{i,j}$ and $\mathbb{E}[\sum_{i=1}^{m} X_i] = 32 \log m$. Using Theorem3.2.2 we have that the probability $|M(j)| = \sum_{i=1}^{m} X_i$ is smaller than $5 \log m$ is at most $\exp(-\frac{32 \log m}{8}) = \frac{1}{m^4}$. Thus $|M(j)| \geq 5 \log m$ with probability at least $1 - \frac{1}{m^4}$.

Consider any machine i. Let G_i be the random variable with value 1 if there is no job $j' \neq j$ such that $i \in M(j')$. Otherwise G_i has value 0. By definition of the algorithm we have the following.

$$\begin{aligned}
\Pr[G_i = 1] &= \Pr[i \notin M(j') \ \ \forall j \neq j'] && (3.2)\\
&= \prod_{j' \neq j} \Pr[i \notin M(j')] && (3.3)\\
&= \prod_{j' \neq j} (1 - 32 \log m \cdot x_{i,j'}) && (3.4)\\
&\geq \exp(-32 \log m \sum_{j' \neq j} x_{i,j'}) && (3.5)\\
&\geq \exp(-32 \log m T(i)) && (3.6)\\
&\geq \frac{1}{e^{1/2}} \quad [T(i) \leq \tfrac{1}{64 \log m} \text{ by assumption}] && (3.7)
\end{aligned}$$

Let E_j denote the event that $|M(j)| \geq 5 \log m$. Consider the probability that $G := \sum_{i \in M(j)} (1 - G_i)$ given that E_j occurs. This is the probability of the bad event where no machine in $M(j)$ is selected only by j given E_j. Given a set $M(j)$, we know $\mathbb{E}[G \mid M(j)] = \mathbb{E}[\sum_{i \in M(j)} G_i \mid M(j)] = \sum_{i \in M(j)} \mathbb{E}[G_i] = \sum_{i \in M(j)} \frac{1}{e^{1/2}} = \frac{|M(j)|}{e^{1/2}}$. This holds for all sets $M(j)$. Thus, $\mathbb{E}[G \mid E_j] \geq \frac{5 \log m}{e^{1/2}}$. Using Theorem 3.2.2 the probability that $G := \sum_{i \in M(j)} (1 - G_i)$ given E_j is at most $\exp(-\frac{5 \log m}{2e^{1/2}}) \geq \frac{1}{m^{3/2}}$.

We now put the above facts together. The probability E_j does not occur is at most $\frac{1}{m^4}$. The probability that $G = 0$ given E_j occurs is at most $\frac{1}{m^{3/2}}$. One of these events must occur for G to be 0. Thus, a union bound says that the probability $G = 0$ is at most $\frac{1}{m^4} + \frac{1}{m^{3/2}} \leq \frac{1}{m}$. Therefore, the probability $G \geq 1$ happens with probability at least $1 - \frac{1}{m}$, implying that there is a machine i in $M(j)$ such that no other job j' has $i \in M(j')$. □

Assigning the Small Jobs

In this section we show how the small jobs can be assigned. For each small job, we preprocess its fractional assignment as follows. First we set $x'_{ij} = 0$ for $i \notin S_j$, then set $x'_{ij} = x_{ij} / \sum_{i' \in S_j} x_{i'j}$ for $i \in S_j$. It is easy to verify that this transformation has the following properties.

- $x'_{ij} \geq 0$ and $\sum_{i \in N(j)} x'_{ij} = 1$
- $x'_{ij} \leq 2x_{ij}$

- If $x'_{ij} > 0$ then $p_{ij} \leq T/\log(m)$

This is all the preprocessing we need to do for small jobs. Afterwards all small jobs are assigned using randomized rounding.

Algorithm 8 Randomized Rounding

```
1: for each job $j$ do
2:     Sample $i \in N(j)$ according to the distribution $\{x_{ij}\}_{i=1}^{m}$
3:     Assign job $j$ to machine $i$
4: end for
```

Note that the preprocessing and the randomized rounding can be executed online.

Lemma 3.5.15. *Let S be the set of all small jobs. Applying randomized rounding with the preprocessed fractional assignments x'_{ij} yields a makespan of $O(T)$ for just the jobs in S with high probability.*

Proof. Let X_{ij} be the indicator random variable for the event that $j \in S$ is assigned to machine i. By definition of randomized rounding, we have that the random variables X_{ij} are independent for varying j. Let $L_i^S = \sum_{j \in S \cap N(i)} p_{ij} X_{ij}$ be the load of the small jobs on machine i. Computing expectations we have

$$\mathbb{E}[L_i^S] = \sum_{j \in S \cap N(i)} p_{ij} x'_{ij} \leq 2 \sum_{j \in S \cap N(i)} p_{ij} x_{ij} \leq 2T.$$

Applying Theorem 3.2.3 with $v = \sum_{j \in S \cap N(i)} p_{ij}^2 x'_{ij} \leq 2 \frac{T}{\log m} \mathbb{E}[L_i^S] \leq \frac{2T^2}{\log m}$, $a \leq \frac{T}{\log m}$, and $\lambda = cT$ for some large enough constant c we have

$$\begin{aligned}
\Pr[L_i^S > 2T + \lambda] &\leq \Pr[L_i^S > \mathbb{E}[L_i^S] + \lambda] \\
&\leq \exp\left(\frac{-\lambda^2}{2v + a\lambda/3}\right) \\
&= \exp\left(\frac{-c^2T^2}{\frac{4T^2}{\log m} + \frac{cT^2/3}{\log m}}\right) \\
&= \exp\left(-d \log m\right)) = m^{-d},
\end{aligned}$$

where $d = \frac{3c^2}{12+c}$ is a constant. Now taking a union bound over all machines, we have $L_i^S \leq (c+2)T$ for all i with probability at least $1 - m^{d-1}$. Since c is some constant, this proves the lemma. □

3.6 Lower Bounds for Online Rounding

In this section we aim to prove the following result, as stated in Section 3.1.

Theorem 3.6.1. *Let x be a fractional assignment of restricted assignment jobs that is received online and let $T := \max\{\max_i \sum_{j \in N(i)} p_j x_{ij}, \max_j p_j\}$ be the adjusted fractional makespan. No deterministic algorithm for converting x to an integer assignment can be $o(\log m / \log\log m)$-competitive with respect to T. Further, no randomized algorithm for the same task can be $o(\log\log m / \log\log\log m)$-competitive with respect to T.*

3.6.1 Deterministic Lower Bound

In this section we give a bad instance for deterministic online rounding algorithms for makespan. A rounding algorithm converts a fractional solution $x_{ij} \geq 0$ in which $\sum_{i \in N(j)} x_{ij} = 1$ for each job j into an assignment of job j on some machine $i \in N(j)$. For a sequence of n jobs with fractional solutions, the fractional makespan is $\max_i \sum_{j \in N(i)} p_j x_{ij}$. Due to bad integrality gaps for some instances, we compare our algorithms to $T := \max\{\max_i \sum_{j \in N(i)} p_j x_{ij}, \max_j p_j\}$, which we refer to as the adjusted fractional makespan. We show that for any deterministic online rounding algorithm there is an instance for which it incurs a large makespan when compared to T.

Lemma 3.6.2. *For any deterministic online rounding algorithm A there exists a sequence of unit size jobs such that A has makespan $\log m / \log\log m$ while the fractional makespan is $1/\log\log m$ and the optimal solution has makespan 1.*

Proof. Fix the deterministic rounding algorithm A. Let λ, p and m be integers such that $\lambda^p = m$. The exact value of λ will be chosen later. We consider an instance with m machines. Each job in our sequence will have a size of 1 and a neighborhood of cardinality λ and fractional solution $x_{ij} = \frac{1}{\lambda}$ for each $i \in N(j)$. The bad sequence of jobs will consist of p phases. In the first phase, we release m/λ jobs, each with *disjoint* neighborhoods of size λ. We observe where A assigns these jobs. Since these jobs had disjoint neighborhoods they get assigned to different machines. Let M' be the set of machines where a job got assigned and recurse on this set of machines, starting a new phase. Note that $|M'| = m/\lambda$. This recursion continues until we run out of machines. By our choice of λ, p, m, there are p phases since $\lambda^p = m$.

Letting $\lambda = \log(m)$, we observe that the rounding algorithm's makespan is $p = \log(m)/\log\log(m)$, while the optimal solution in hindsight has makespan 1. It is also easy to verify that the fractional makespan is $p/\lambda = 1/\log\log(m)$. □

For the above sequence $T = \max\{\frac{1}{\log\log m}, 1\}$, and so this implies the $\Omega(\frac{\log m}{\log\log m})$ lower bound for deterministic algorithms. Note that using a uniform fractional assignment on other sequences of jobs such as the one described in [21] does not suffice. For an analysis of a deterministic algorithm on these sequences of jobs we would have $T = \Theta(\log m)$, while the algorithm's makespan would be $\Omega(\log m)$. Thus the resulting ratio would be constant.

3.6.2 Randomized Lower Bound

Applying Yao's principle [152], we aim to give a distribution over instances such that any deterministic algorithm A has a large makespan in expectation when compared to the corresponding fractional makespan. Lemma 3.6.2 implies that for each algorithm A, there exists an instance I_A for which A has makespan at least $\Omega(\log m/\log\log m)$ factor bigger than the corresponding value of T for the instance. We start by describing a distribution for instances on m machines. Afterwards we boost this to a distribution for instances on $M := mk$ machines for some parameter k. We conclude the lower bound by analyzing the resulting makespan in terms of M.

Distribution for instances on m machines

As hinted above, our distribution over instances on m machines will be uniform over all possible instances described in Lemma 3.6.2. Fix integers λ, p and m such that $\lambda^p = m$. In particular we use $\lambda = \log m$ and $p = \log m/\log\log m$ as in Lemma 3.6.2. Let $\mathcal{I}$ be the set of all instances of the form given by Lemma 3.6.2 with parameters λ, p and m. Then our distribution over instances is uniform over $\mathcal{I}$, i.e. for any instance I we set

$$\Pr[\text{send } I] = \begin{cases} 1/|\mathcal{I}| & \text{if } I \in \mathcal{I} \\ 0 & \text{otherwise} \end{cases}$$

We now analyze this distribution and state some key properties it has.

Proposition 3.6.3. *The set of instances $\mathcal{I}$ has the following properties:*

1. $|\mathcal{I}| \leq O\left(\lambda^{O(p^2\lambda^p)}\right)$
2. $|\mathcal{I}| \geq \Omega(\lambda^{\Omega(p\lambda^p)})$
3. *For every $I \in \mathcal{I}$, the corresponding fractional makespan is* $1/\log\log m$
4. *For every deterministic algorithm A, there exists $I_A \in \mathcal{I}$ such that A has makespan at least* $\log m/\log\log m$.

Proof. The last two points follow from Lemma 3.6.2, so we prove the first two points. To bound the number of such instances we describe a process to generate an instance of $\mathcal{I}$. We start by choosing a set of λ machines from the set of m machines, then λ from the remaining $m - \lambda$ machines, and so on. Each set corresponds to unit size job with the set of machines as its neighborhood. Afterwards, we choose of m/λ machines, one from each set, and recurse on these machines. We count the number of ways to pick the initial set of jobs as

$$\binom{m}{\lambda}\binom{m-\lambda}{\lambda}\cdots\binom{m-(m/\lambda)\lambda}{\lambda} = \frac{m!}{(\lambda!)^{m/\lambda}}$$

To see this equality note that corresponding terms in the numerators and denominators cancel out, leaving just the first $m!$ and a $\lambda!$ for each binomial term. After picking these jobs, there are $\lambda^{m/\lambda}$ ways to choose the set of machines to recurse on since there are m/λ

sets of size λ and we choose one machine from each. Applying this idea recursively, we get that

$$|\mathcal{I}| = \prod_{\ell=0}^{p} \frac{(m/\lambda^{\ell})!}{(\lambda!)^{m/\lambda^{\ell}}} \lambda^{m/\lambda^{\ell}}$$

At some loss, we upper bound this by taking the first term (since it's the largest) in the product above to the p'th power.

$$|\mathcal{I}| \leq \left(\frac{m!}{(\lambda!)^{m/\lambda}} \lambda^{m/\lambda} \right)^{p}$$

We now use the fact that $m = \lambda^p$ to express everything in terms of only λ and p.

$$|\mathcal{I}| \leq \left(\frac{(\lambda^p)!}{(\lambda!)^{\lambda^{p-1}}} \lambda^{\lambda^{p-1}} \right)^{p}$$

Using Sterling's approximation for factorial we have that $(\lambda^p)! = O(\lambda^{p(\lambda^p+1)})$ and $\lambda! \geq \Omega(\lambda^{\lambda})$. Now combining these two inequalities we have that

$$|\mathcal{I}| \leq O\left(\lambda^{O(p^2 \lambda^p)} \right)$$

Now to get the lower bound we look at the first term in the product above and substitute $m = \lambda^p$.

$$|\mathcal{I}| \geq \frac{m!}{(\lambda!)^{m/\lambda}} \lambda^{m/\lambda} = \frac{(\lambda^p)!}{(\lambda!)^{\lambda^{p-1}}} \lambda^{\lambda^{p-1}}$$

Again using Sterling's approximation for factorial we have $(\lambda^p)! = \Omega(\lambda^{p\lambda^p})$ and $\lambda! = O(\lambda^{\lambda+1})$. Combining these yields

$$|\mathcal{I}| \geq \Omega(\lambda^{\Omega(p\lambda^p)})$$

completing the proof. □

The above proposition implies that for any deterministic algorithm A, the probability that A incurs makespan at least $\Omega(\log m / \log\log m)$ is $1/|\mathcal{I}| \geq \Omega(1/\lambda^{O(p^2\lambda^p)})$

Boosting the Distribution

Since the above distribution on m machines has a low probability of incurring a high makespan on some deterministic algorithm A, we need to boost this in order to conclude our lower bound. Let $k := |\mathcal{I}|$ and set $M := mk$. We construct a distribution for instances on M machines as follows. Partition the set of M machines into k groups of m machines and on each group independently sample an instance from $\mathcal{I}$.

Let $I_1, I_2, \ldots, I_k$ be the sampled instances on each group of machines. For any deterministic algorithm A we have that group s has makespan $\log m / \log\log m$ if $I_s = I_A$, which happens with probability at least $1/k$. Using this we can show the following lower bound on the expected makespan of algorithm A.

Lemma 3.6.4. *The expected makespan of any deterministic algorithm A on the above distribution over instances on M machines is at least $\Omega(\log m/\log\log m)$.*

Proof. Algorithm A has makespan $\Omega(\log m/\log\log m)$ if any group's sampled instance is equal to A's bad instance, I_A. This occurs with the following probability:

$$\begin{aligned}\Pr[\vee_{s=1}^{k}(I_s = I_A)] &= 1 - \Pr[\wedge_{s=1}^{k}(I_s \neq I_A)] \\ &= 1 - \prod_{s=1}^{k}(1 - \Pr[I_s = I_k]) \\ &\geq 1 - (1-1/k)^k \\ &\geq 1 - 1/e = \Omega(1)\end{aligned}$$

So A's expected makespan is $\Omega(\log m/\log\log m)$. □

Finally, we just need to conclude that $\log\log M = O(\log m)$ and $\log\log\log M = \Omega(\log\log m)$ to finish the proof of the lower bound.

Lemma 3.6.5. *For $M := mk$, we have $\log\log(M) = O(\log m)$ and $\log\log\log M = \Omega(\log\log m)$*

Proof. Since $k = O\left(\lambda^{O(p^2\lambda^p)}\right)$ and $m = \lambda^p$, we have $M = O\left(\lambda^{O(p^2\lambda^p)}\right)$ as well. Thus we have $\log M = O\left(p^2\lambda^p \log\lambda\right)$ and

$$\log\log M = O(\log p + p\log\lambda + \log\log\lambda) = O(p\log\lambda).$$

Now using that $p = \log m/\log\log m$ and $\lambda = \log m$ we have that $\log\log M = O(\log m)$.

Similarly, since $k = \Omega(\lambda^{\Omega(p\lambda^p)})$ and $m = \lambda^p$, we have $M = mk = \Omega(\lambda^{\Omega(p\lambda^p)})$. Thus we have $\log M = \Omega(p\lambda^p\log\lambda)$ and $\log\log M = \Omega(\log p + p\log\lambda + \log\log\lambda)$. Thus we have

$$\log\log\log M = \Omega(\log p) = \Omega(\log\log m)$$

since $p = \log m/\log\log m$. □

Finally, we see that Lemmas 3.6.2, 3.6.4, and 3.6.5 imply Theorem 3.6.1.

3.6.3 Learning the Weights

We show that machine weights for makespan minimization are learnable from data in the following formal sense. There is an unknown distribution $\mathcal{D}$ over instances of the problem. A sample $S \sim \mathcal{D}$ consists of n jobs, where job j has size p_j and neighborhood $N(j) \subseteq [m]$ of machines. For simplicity, we assume $\mathcal{D} = \prod_{j=1}^{n}\mathcal{D}_j$, i.e. each job is sampled independently from it's own "private" distribution and that $p_j = 1$ for all jobs. Later we show how to generalize to different sizes. Let $\mathrm{ALG}(w, S)$ be the fractional makespan on instance S with weights w. We want to show that we can find weights w given s samples $S_1, S_2, \ldots, S_s$ from $\mathcal{D}$ such that $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w, S)] \leq (1 + O(\epsilon))\mathbb{E}[\mathrm{OPT}(S)]$ with high probability (i.e. probability at least $1-\delta$ for $\delta > 0$. Here $\mathrm{OPT}(S)$ is the optimal (fractional) makespan on job set S. Note that such a result also implies that these weights w also satisfy $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w,S)] \le (1+O(\epsilon))\min_{w'}\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w',S)]$ with high probability, i.e. they are comparable to the best set of weights for the distribution $\mathcal{D}$. Ideally, we want $s = \mathrm{poly}(m, \frac{1}{\epsilon}, \frac{1}{\delta})$ number of samples, and lower is better.

Preliminary Results on Proportional Weights

We need the following prior results about the weights. Recall that given a set of jobs S and weights $w \in \mathbb{R}^m_+$ we consider the following fractional assignment rule for job j and $i \in N(j)$.

$$x_{ij}(w) := \frac{w_i}{\sum_{i'\in N(j)} w_{i'}} \tag{3.8}$$

For ease of notation we assume that $x_{ij} = 0$ whenever $i \notin N(j)$. We would like to find weights w such that $x_{ij}(w)$ approximately solves the following LP.

$$\begin{aligned} \text{maximize} \quad & \sum_i \sum_j x_{ij} \\ & \sum_j x_{ij} \le T_i \quad \forall i \in [m] \\ & \sum_i x_{ij} \le 1 \quad\ \ \forall j \in S \\ & x \ge 0 \end{aligned} \tag{3.9}$$

Here, the right hand side values T_i are inputs and can be thought as all being set to the optimal makespan. Given an assignment via the weights $x_{ij}(w)$ via weights w, we can always convert it to a feasible solution to LP (3.9) in the following way. For all $i \in [m]$ let $O_i = \max\{\sum_j x_{ij}(w)/T_i, 1\}$. It is easy to see that x' is feasible for LP (3.9) and that the amount lost is exactly the overallocation $\sum_i \max\{\sum_j x_{ij}(w) - T_i, 0\}$. We can then take $x'_{ij} = x_{ij}(w)/O_i$ for all i, j. The following theorem is adapted from Agrawal et al.

Theorem 3.6.6 (Theorem 1 in Agrawal et al.). *For any $\delta \in (0,1)$, there exists an algorithm which finds weights w such that a downscaling of $x_{ij}(w)$ is a $1-\delta$-approximation to LP (3.9). The algorithm operates in $R = O(\frac{1}{\delta^2}\log(m/\delta))$ iterations and produces weights of the form $w_i = (1+\epsilon)^k$ for $k \in [0, R]$.*

Using this theorem, we get the following result as simple corollary. Again let S be set of n jobs that we want to schedule on m machines to minimize the makespan. Let T be the makespan of an optimal schedule

Corollary 3.6.7. *For any $\epsilon > 0$, there exists weights $w \in \mathbb{R}^m_+$ such that $x_{ij}(w)$ yields a fractional schedule with makespan at most $(1+\epsilon)T$. The weights are computed by running for $R = O(m^2\log(m/\epsilon)/\epsilon^2$ iterations and produces weights of the form $w_i = (1+\epsilon)^k$ for $k \in [0, R]$.*

Proof. Consider running the algorithm of Theorem 3.6.6 with $\delta = \epsilon/m$ and $T_i = T$ for all i. The optimal value of (3.9) on this instance is exactly n since T is the optimal makespan and thus we are able to assign all the jobs. After scaling down to be feasible, the solution has value at least $(1 - \epsilon/m)n$. We only scaled down the assignment on machines for which its assignment was greater than T, and the amount we lost in this scaling down was at most $\epsilon n/m$. Thus in the worst case, any machines assignment using the weights is at most $T + \epsilon n/m \leq (1 + \epsilon)T$, since $T \geq n/m$. □

Learning the Weights by Stacking

Our learning algorithm will be to compute weights on a "stacked" instance, that is we will aggregate all of the instances together into a single large instance. Our goal for this section will be to show that this is a reasonable thing to do. Let's set up some more notation. Let $\mathcal{W}(R)$ be the set of possible weights output by R iterations of the proportional algorithm. Let $\mathrm{OPT}(S)$ be the optimal fractional makespan on job set S. We are interested in the case when $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{OPT}(S)] = \Omega(\log m)$. Let $L_i(w, S)$ be the fractional load of machine i on instance S with weights w. Thus we have $\mathrm{ALG}(w, S) = \max_i L_i(w, S)$. Note that $L_i(w, S) = \sum_{j\in S} x_{ij}(w)$. Our first lemma shows that $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w, S)] \approx \max_i \mathbb{E}_{S\sim\mathcal{D}}[L_i(w, S)]$. When it is clear, we will suppress $S \sim \mathcal{D}$ for ease of notation.

Lemma 3.6.8. *Let $\epsilon > 0$ be given. If $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{OPT}(S)] \geq \frac{4+2\epsilon}{\epsilon^2}\log(\frac{m}{\sqrt{\epsilon}})$, then for all R and all weights $w \in \mathcal{W}(R)$, we have $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w, S)] \leq (1 + 2\epsilon)\max_i \mathbb{E}_{S\sim\mathcal{D}}[L_i(w, S)]$.*

Proof. Fix any R and $w \in \mathcal{W}(R)$. We have the following simply bound on $\mathrm{ALG}(w, S)$. It can either be at most $(1 + \epsilon)\max_i \mathbb{E}[L_i(w, S)]$, or it is larger in which case it is at most n. Thus we have:

$$\begin{aligned}\mathbb{E}[\mathrm{ALG}(w, S)] \leq& (1 + \epsilon)\max_i \mathbb{E}[L_i(w, S)] \\ &+ n\Pr[\mathrm{ALG}(w, S) > (1 + \epsilon)\max_i \mathbb{E}[L_i(w, S)]] \\ \leq& (1 + \epsilon)\max_i \mathbb{E}[L_i(w, S)] + n\sum_i \Pr[L_i(w, S) \geq (1 + \epsilon)\mathbb{E}[L_i(w, S)]]\end{aligned}$$

Now we claim that for each i, $\Pr[L_i(w, S) \geq (1 + \epsilon)\mathbb{E}[L_i(w, S)]] \leq \epsilon/m^2$. Indeed, if this is the case then we see that

$$\mathbb{E}[\mathrm{ALG}(w, S)] \leq (1 + \epsilon)\max_i \mathbb{E}[L_i(w, S)] + \frac{\epsilon n}{m} \leq (1 + 2\epsilon)\max_i \mathbb{E}[L_i(w, S)]$$

since $\max_i \mathbb{E}[L_i(w, S)] \geq n/m$, and thus proving the lemma. Thus we just need to show the claim. Recall that $L_i(w, S) = \sum_{j\in S} x_{ij}(w)$ and that each job j is chosen to be part of S independently from distribution $\mathcal{D}_j$. Thus $x_{ij}(w)$ is an independent random variable in the interval $[0, 1]$ for each j. Applying Theorem 3.2.1 to $L_i(w, S)$ with $\mu = \max_{i'} \mathbb{E}[L_{i'}(w, S)]$, we see that since $\mu \geq \mathbb{E}[\mathrm{OPT}(S)] \geq \frac{4+2\epsilon}{\epsilon^2}\log(\frac{m}{\sqrt{\epsilon}})$, we have

$$\Pr[L_i(w, S) > (1 + \epsilon)\mu] \leq \exp\left(-\frac{\epsilon^2}{2 + \epsilon}\mu\right) \leq \exp\left(-\frac{\epsilon^2}{2 + \epsilon}\mathbb{E}[\mathrm{OPT}(S)]\right) \leq \frac{\epsilon}{m^2}$$

completing the proof of the claim. □

Now that we have this lemma, we can show that computing the weights on a "stacked" instance suffices to find weights that generalize for the distribution. The result we want to prove is the following.

Theorem 3.6.9. *Let $\epsilon, \delta \in (0,1)$ and $R = O(\frac{m^2}{\epsilon^2}\log(\frac{m}{\epsilon}))$ be given and let $\mathcal{D} = \prod_{j=1}^n \mathcal{D}_j$ be a distribution over n-job restricted assignment instances such that $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{OPT}(S)] \geq \Omega(\frac{1}{\epsilon^2}\log(\frac{m}{\epsilon}))$. There exists an algorithm which finds weights $w \in \mathcal{W}(R)$ such that $\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w,S)] \leq (1+O(\epsilon))\min_{w'\in\mathcal{W}(R)}\mathbb{E}_{S\sim\mathcal{D}}[\mathrm{ALG}(w',S)]$ when given access to $s = \mathrm{poly}(m, \frac{1}{\epsilon}, \frac{1}{\delta})$ independent samples $S_1, S_2, \dots, S_s \sim \mathcal{D}$. The algorithm succeeds with probability at least $1 - O(\delta)$ over the random choice of samples.*

We will show that *uniform convergence* occurs when we take $s = \mathrm{poly}(m, \frac{1}{\epsilon}, \frac{1}{\delta})$ samples. This means that for all $i \in [m]$ and all $w \in \mathcal{W}(R)$ simultaneously, we have with probability $1-\delta$ that $\frac{1}{s}\sum_\alpha L_i(w, S_\alpha) \approx \mathbb{E}_S[L_i(w,S)]$. Intuitively this should happen because the class of weights $\mathcal{W}(R)$ is not too complex. Indeed we have that $|\mathcal{W}(R)| = R^m$, and thus the pseudo-dimension is $\log(|\mathcal{W}(R)|) = m\log(R) = O(m\log m)$ when $R = O(m^2\log m)$. Once we have established uniform convergence, setting up the algorithm and analyzing it will be quite simple. We start with some lemmas showing uniform convergence.

Lemma 3.6.10. *Let $\epsilon, \delta \in (0,1)$ and $S_1, S_2, \dots, S_s \sim \mathcal{D}$ be independent samples. If $s \geq \frac{m^2}{\epsilon^2}\log(\frac{2|\mathcal{W}(R)|m}{\delta})$, then with probability at least $1-\delta$ for all $i \in [m]$ and $w \in \mathcal{W}(R)$ we have*

$$\left|\frac{1}{s}\sum_\alpha L_i(w,S_\alpha) - \mathbb{E}_S[L_i(w,S)]\right| \leq \epsilon \max_{i'} \mathbb{E}_S[L_{i'}(w,S)]$$

Proof. Fix a machine $i \in [m]$ and $w \in \mathcal{W}(R)$. We have that $\frac{1}{s}L_i(w,S_\alpha)$ is an independent random variable in $[0, n/s]$ for each $\alpha \in [s]$. Moreover we have that $\mathbb{E}[\frac{1}{s}\sum_\alpha L_i(w,S_\alpha)] = \mathbb{E}[L_i(w,S)]$. Applying Theorem 3.2.4 to $\frac{1}{s}\sum_\alpha L_i(w,S_\alpha)$ with $t = \epsilon\max_{i'}\mathbb{E}[L_{i'}(w,S)$, we have

$$\Pr\left[\left|\frac{1}{s}\sum_\alpha L_i(w,S_\alpha) - \mathbb{E}[L_i(w,S)]\right| \geq t\right] \leq 2\exp\left(-s\frac{\epsilon^2(\max_{i'}\mathbb{E}[L_{i'}(w,S)])^2}{n^2}\right).$$

We claim that if $s \geq \frac{m^2}{\epsilon^2}\log(\frac{2|\mathcal{W}(R)|m}{\delta})$, then this probability is at most $\frac{\delta}{|\mathcal{W}(R)|m}$. Indeed, this claim follows if $m \geq \frac{n}{\max_{i'}\mathbb{E}[L_{i'}(w,S)]}$, which is true since $\max_{i'}\mathbb{E}[L_{i'}(w,S)] \geq n/m$. Finally, the lemma follows by union bounding over all $i \in [m]$ and $w \in \mathcal{W}(R)$. □

Lemma 3.6.11. *Let $\epsilon, \delta \in (0,1)$ and $S_1, S_2, \dots, S_s \sim \mathcal{D}$ be independent samples. For each $\alpha \in [s]$ let $T_\alpha = \mathrm{OPT}(S_\alpha)$. If $s \geq \frac{m^2}{\epsilon^2}\log(2/\delta)$ then with probability at least $1-\delta$ we have*

$$(1-\epsilon)\mathbb{E}[\mathrm{OPT}(S)] \leq \frac{1}{s}\sum_\alpha T_\alpha \leq (1+\epsilon)\mathbb{E}[\mathrm{OPT}(S)]$$

Proof. For each $\alpha \in [s]$ we have $\frac{1}{s}T_\alpha$ is an independent random variable in $[0, n/s]$. Moreover, we have $\mathbb{E}[\frac{1}{s}\sum_\alpha T_\alpha] = \mathbb{E}[\mathrm{OPT}(S)]$. Applying Theorem 3.2.4 to $\frac{1}{s}\sum_\alpha T_\alpha$ we have

$$\Pr\left[|\frac{1}{s}\sum_\alpha T_\alpha - \mathbb{E}[\mathrm{OPT}(S)]| \geq \epsilon \mathbb{E}[\mathrm{OPT}(S)]\right] \leq 2\exp\left(-s\frac{\epsilon^2\mathbb{E}[\mathrm{OPT}(S)]^2}{n^2}\right).$$

Now since $\mathbb{E}[\mathrm{OPT}(S)] \geq n/m$ we have this probability is at most $2\exp(-s\epsilon^2/m^2)$. Thus whenever $s \geq \frac{m^2}{\epsilon^2}\log(2/\delta)$, this probability becomes at most δ, completing the proof. □

Analyzing the Learning Algorithm

We can now formally describe and analyze the algorithm. Set $R = O(\frac{m^2}{\epsilon^2}\log(m/\epsilon)$. We sample independent instances $S_1, S_2, \ldots, S_s \sim \mathcal{D}$ for $s \geq \frac{m^2}{\epsilon^2}\log(\frac{2|\mathcal{W}(R)|m}{\delta})$ Next we set up a stacked instance consisting of all the jobs in these samples. Next we set $T_\alpha = \mathrm{OPT}(S_\alpha)$ and $T = \sum_\alpha T_\alpha$. We run the algorithm of Corollary 3.6.7 on the stacked instance with right hand side bounds $T_i = T$ for all i. The algorithm should run for R rounds and produce weights $w \in \mathcal{W}(R)$ such that $\sum_\alpha L_i(w, S_\alpha) \leq (1+\epsilon)T$ for all i. We can now prove Theorem 3.6.9.

Proof of **Theorem 3.6.9**. Let $w \in \mathcal{W}(R)$ be the weights output by the algorithm above. Now for a new randomly sampled instance $S \sim \mathcal{D}$, by Lemma 3.6.8 we have that

$$\mathbb{E}[\mathrm{ALG}(w, S)] \leq (1+2\epsilon)\max_i \mathbb{E}[L_i(w, S)].$$

By Lemma 3.6.10, we have $\max_i \mathbb{E}[L_i(w, S)] \leq (1+O(\epsilon))\max_i \frac{1}{s}\sum_\alpha L_i(w, S_\alpha)$ with probability at least $1-\delta$. By construction of our algorithm, we have $\sum_\alpha L_i(w, S_\alpha) \leq (1+\epsilon)T = (1+\epsilon)\sum_\alpha T_\alpha$ for all i. It thus follows that $\max_i \mathbb{E}[L_i(w, S)] \leq (1+O(\epsilon))\frac{1}{s}\sum_\alpha T_\alpha$ with probability at least $1-\delta$. Next we have that $\frac{1}{s}\sum_\alpha T_\alpha \leq (1+\epsilon)\mathbb{E}[\mathrm{OPT}(S)]$ with probability at least $1-\delta$. Finally, with probability at least $1-2\delta$, by chaining these inequalities together we get

$$\mathbb{E}[\mathrm{ALG}(w, S)] \leq (1+O(\epsilon))\mathbb{E}[\mathrm{OPT}(S)] \leq (1+O(\epsilon))\mathbb{E}[\mathrm{ALG}(w^*, S)]$$

where $w^* = \arg\min_{w' \in \mathcal{W}(R)} \mathbb{E}[\mathrm{ALG}(w^*, S)]$. Since $R = O(\frac{m^2}{\epsilon^2}\log(\frac{m}{\epsilon}))$, we have that $\log(|\mathcal{W}(R)|) = m\log(R) = O(m\log(m/\epsilon))$. Thus we can take $s = \mathrm{poly}(m, \frac{1}{\epsilon}, \frac{1}{\delta})$ to get the result. This completes the proof. □

Handling Different Sizes

Now we give a sketch of how to handle the case when each job has an integer size $p_j > 0$. For this we need a slightly different version of Theorem 3.6.6 and Corollary 3.6.7. Consider the following variant of LP 3.9:

$$\begin{aligned}\text{maximize} \quad & \sum_j p_j \sum_i x_{ij} \\ & \sum_j p_j x_{ij} \le T_i \quad \forall i \in [m] \\ & \sum_i x_{ij} \le 1 \qquad \forall j \in S \\ & x \ge 0\end{aligned} \tag{3.10}$$

Again we simplify notation and assume $x_{ij} = 0$ whenever $i \notin N(j)$. The following is a corollary of Theorem 3.6.6. Let T be the optimal makespan for a set of jobs

Corollary 3.6.12. *For any $\epsilon > 0$, there exists weights $w \in \mathbb{R}^m_+$ such that $x_{ij}(w)$ yields a fractional schedule with makespan at most $(1+\epsilon)T$. The weights are computed by running a variant of the algorithm of Theorem 3.6.6 for $R = O(m^2 \log(m/\epsilon)/\epsilon^2$ iterations and produces weights of the form $w_i = (1+\epsilon)^k$ for $k \in [-R, R]$.*

Proof. Consider creating p_j unit-sized copies of each job j. Note that this only needs to be done conceptually. It is easy to see that writing down LP 3.9 for this conceptual instance is a relaxation of LP 3.10. Consider running the Algorithm of Theorem 3.6.6 with $\delta = \epsilon/m$, $T_i = T$ for all $i \in [m]$ and for $R = O(\frac{1}{\delta^2}\log(fracm\delta)$ iterations. Note that since T is the optimal makespan, there exists a solution with value $\sum_j p_j$. Thus since the algorithm returns a $(1-\delta)$-approximation, we get a solution with value at least $(1-\delta)\sum_j p_j$. The amount that we lose in the objective is exactly the total amount over-allocated in the solution given by the weights. Thus for all i, since $T \ge \sum_j p_j/m$. we have

$$\sum_j p_j x_{ij}(w) \le T + \delta \sum_j p_j = T + \epsilon \frac{\sum_j p_j}{m} \le (1+\epsilon)T.$$

□

Our learning algorithm will be the same as before, just the jobs will now have sizes. We go through each lemma above and prove an analogous version for when there are job sizes. For a job set S and weights $w \in \mathcal{W}(R)$ let $L_i(w, S) = \sum_j p_j x_{ij}(w)$. Let $p_{\max} = \max_j p_j$ be the maximum job size. For this case our assumption becomes $\mathbb{E}_{S\sim\mathcal{D}}[\text{OPT}(S)] \ge \frac{4+2\epsilon}{\epsilon^2} p_{\max} \log(\frac{m}{\sqrt{\epsilon}})$.

Lemma 3.6.13. *Let $\epsilon > 0$ be given. If $\mathbb{E}_{S\sim\mathcal{D}}[\text{OPT}(S)] \ge \frac{4+2\epsilon}{\epsilon^2} p_{\max} \log(\frac{m}{\sqrt{\epsilon}})$, then for all R and all weights $w \in \mathcal{W}(R)$, we have $\mathbb{E}_{S\sim\mathcal{D}}[\text{ALG}(w, S)] \le (1+2\epsilon)\max_i \mathbb{E}_{S\sim\mathcal{D}}[L_i(w, S)]$.*

Proof. Fix any R and $w \in \mathcal{W}(R)$. We have the following simply bound on ALG(w, S). It can either be at most $(1+\epsilon)\max_i \mathbb{E}[L_i(w, S)]$, or it is larger in which case it is at most

$\sum_j p_j$. Thus we have:

$$\begin{aligned}\mathbb{E}[\mathrm{ALG}(w,S)] \leq& (1+\epsilon)\max_i \mathbb{E}[L_i(w,S)] \\ &+ \sum_j p_j \Pr[\mathrm{ALG}(w,S) > (1+\epsilon)\max_i \mathbb{E}[L_i(w,S)]] \\ \leq& (1+\epsilon)\max_i \mathbb{E}[L_i(w,S)] + \sum_j p_j \sum_i \Pr[L_i(w,S) \geq (1+\epsilon)\mathbb{E}[L_i(w,S)]]\end{aligned}$$

Now we claim that for each i, $\Pr[L_i(w,S) \geq (1+\epsilon)\mathbb{E}[L_i(w,S)]] \leq \epsilon/m^2$. Indeed, if this is the case then we see that

$$\mathbb{E}[\mathrm{ALG}(w,S)] \leq (1+\epsilon)\max_i \mathbb{E}[L_i(w,S)] + \frac{\epsilon\sum_j p_j}{m} \leq (1+2\epsilon)\max_i \mathbb{E}[L_i(w,S)]$$

since $\max_i \mathbb{E}[L_i(w,S)] \geq \sum_j p_j/m$, and thus proving the lemma. Thus we just need to show the claim. Recall that $L_i(w,S) = \sum_{j\in S} p_j x_{ij}(w)$ and that each job j is chosen to be part of S independently from distribution $\mathcal{D}_j$. Thus $p_j x_{ij}(w)$ is an independent random variable in the interval $[0, p_{\max}]$ for each j. Applying Theorem 3.2.1 to $L_i(w,S)/p_{\max}$ with $\mu = \max_{i'} \mathbb{E}[L_{i'}(w,S)]/p_{\max}$, we see that since $\mu \geq \mathbb{E}[\mathrm{OPT}(S)]/p_{\max} \geq \frac{4+2\epsilon}{\epsilon^2}\log(\frac{m}{\sqrt{\epsilon}})$, we have

$$\Pr\left[\frac{L_i(w,S)}{p_{\max}} > (1+\epsilon)\mu\right] \leq \exp\left(-\frac{\epsilon^2}{2+\epsilon}\mu\right) \leq \exp\left(-\frac{\epsilon^2}{2+\epsilon}\mathbb{E}[\mathrm{OPT}(S)]\right) \leq \frac{\epsilon}{m^2}$$

which implies the claim. □

Modifying the remaining lemmas is simple. We can do this by replacing most instances of n in the proofs with $\sum_j p_j$.

3.7 Existence of Weights for a Near Optimal Fractional Assignment

We first establish that perhaps surprisingly there exists weights that result in a near optimal fractional assignment using the rule described in (3.1). We build on the recent work of Agrawal et al [5] which shows the existence of such weights for a related problem, and an efficient algorithm to construct the weights given an offline instance. We build upon this work for our objective of makespan minimization on unrelated machines. Their work is concerned with finding approximate solutions to the maximum cardinality bipartite b-matching problem, which is described below in (3.11).

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} \sum_{i \in N(j)} x_{ij} \\ \text{s.t.} \quad & \sum_{j \in N(i)} x_{ij} \leq T_i \quad \forall i \in [m] \\ & \sum_{i \in N(j)} x_{ij} \leq 1 \quad \forall j \in [n] \\ & x \geq 0 \end{aligned} \tag{3.11}$$

We will refer to the first set of constraints as the makespan constraints. Agrawal et al show that there exist weights w such that the proportional assignment $x_{ij}(w)$ is a $(1-\delta)$-approximation to the above problem for any $\delta \in (0,1)$. This is stated formally in the following theorem.

Theorem 3.7.1 (Theorem 1 in [5]). *For any constant $\delta \in (0,1)$ there exists an algorithm that computes weights $w \in \mathbb{R}_+^m$ such that an appropriately scaled down allocation given by (3.1) is feasible and a $(1-\delta)$-approximation for (3.11). The algorithm runs in* $\text{poly}(n, m, \frac{1}{\delta})$ *time.*

Unfortunately, we cannot use this theorem to directly infer the existence of good weights for our problem. First, the above problem only captures the case of restricted assignment with unit size jobs, rather than more general sizes (although this will be easy to fix). Second, the theorem implicitly reduces the assignment of some jobs so that the constraint $\sum_{i \in N(j)} x_{ij} \leq 1$ is no longer tight, despite the fact that assigning via equation (3.1) maintains that this constraint is tight. This reduction is done to ensure strict feasibility for the matching problem, but causes issues for the makespan objective. Indeed, a key difference between the two problems is that for makespan minimization, all jobs must be assigned. Whereas, for approximate b-matchings some jobs that are difficult to assign can be dropped.

Consider the following modified problem that accounts for job sizes. We will consider the case where $T_i := T$ for all $i \in [m]$, since this corresponds to our makespan minimization problem.

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} \sum_{i \in N(j)} x_{ij} \\ \text{s.t.} \quad & \sum_{j \in N(i)} p_j x_{ij} \leq T \quad \forall i \in [m] \\ & \sum_{i \in N(j)} x_{ij} \leq 1 \quad \forall j \in [n] \\ & x \geq 0 \end{aligned} \tag{3.12}$$

We extend the algorithm and analysis of Agrawal et al to this problem. Note that the objective is maximizing the number of assigned jobs and that the objective is n if all jobs can be feasibly assigned. Algorithm 9 describes the procedure for constructing the weights w and is a modification of the algorithm in [5]. The overall procedure is similar to the

Multiplicative Weights Update method [17]. This algorithm takes a parameter $\epsilon \in (0,1)$ as input. The algorithm starts by initializing all w_i to 1, then it updates the weights in successive rounds. In each round we compute the fractional load of each machine, L_i. If L_i greatly violates its makespan constraint, then we should decrease w_i. Similarly, if L_i is very slack for its makespan constraint then w_i should be increased. The increases and decreases are done in small amounts by multiplying or dividing by $1+\epsilon$. Taking $\epsilon = \delta/5$ suffices to yield Theorem 3.7.1 as shown in [5].

Algorithm 9 Proportional Allocation Algorithm

```
1: Input: $\epsilon \in (0,1)$, $R \in \mathbb{Z}_+$, sets $N(i)$, $N(j)$ for each $i, j$, makespan bound $T$
2: for $i = 1, \ldots, m$ do
3:     $w_i \leftarrow 1$                                  ▷ Initialization
4: end for
5: for $r = 1, \ldots, R$ do
6:     for $i = 1, \ldots, m$ do
7:         for $j \in N(i)$ do
8:             Compute $x_{ij}(w)$ as in (3.1)
9:         end for
10:        Compute $L_i = \sum_{j \in N(i)} p_j x_{ij}$
11:        if $L_i \geq (1+\epsilon)T$ then              ▷ Over-allocation
12:            $w'_i \leftarrow w_i/(1+\epsilon)$
13:        else if $L_i \leq T/(1+\epsilon)$ then        ▷ Under-allocation
14:            $w'_i \leftarrow (1+\epsilon)w_i$
15:        end if
16:    end for
17:    $w \leftarrow w'$
18: end for
```

Theorem 3.7.2. *Consider an instance of (3.12) in which there exists an assignment that fully assigns all jobs and satisfies all makespan constraints. Let $n' = \sum_j p_j$. Then the fractional makespan of the assignment output by Algorithm 9 with $\epsilon = \frac{\delta}{5} = \frac{c}{5m}$ is at most $(1+c)T$ for any $c > 0$.*

Proof. We start by noting that we can reduce the restricted assignment problem with job sizes to the unit size case in the following way. For each job j with size p_j, release p_j unit size jobs all with neighborhood $N(j)$. Any solution to (3.11) on the reduced instance translates to a solution to (3.12), in particular the solution generated by the machine weights translates between the problems. Under the assumption that $p_j \leq \text{poly}(m,n)$ this reduction can be done in polynomial time. We note that the number of jobs in this reduced instance is $n' = \sum_{j=1}^{n} p_j$.

Let OPT be the value of an optimal solution to (3.1) on the reduced instance. Since there is a way to assign all jobs and satisfy the makespan constraints, we have that $OPT = n'$. Note that at the end of the algorithm if we assign the variables x_{ij} using (3.1) the objective value is n' since every job is fully allocated. Now from Theorem 3.7.1 we know that the solution after scaling down has value at least $(1-\delta)\text{OPT} = (1-\delta)n'$. The scaling down only occurs on machines for which the makespan constraint is violated. Thus the amount we lost in the objective after scaling is equal to the total amount we have over-assigned the machines. The amount lost is at most $\delta n'$, so the total over-assignment across all machines is at most $\delta n'$. In the worst case, this over-assignment occurs on a single machine. Thus our choice of δ guarantees that the assignment on each machine (before scaling down) is at most

$$T + \delta n' = T + \frac{cn'}{m} \leq (1+c)T.$$

The last inequality follows from the fact that there exists a feasible assignment with makespan T and the pigeonhole principle, so we have that $\frac{n'}{m} \leq T$. □

This establishes that for any offline instance of our problem, there exists $w \in \mathbb{R}_+^m$ such that the allocation given by (3.1) completely assigns all jobs and is a $(1+c)$-approximation for the makespan for any $c > 0$.

3.8 Proof of Rounding Theorem

In this section we combine the results of the analysis in Section 3.5 to conclude Theorem 3.5.1. We recall the statement of this result here.

Theorem 3.8.1 (Theorem 3.5.1 restated)**.** *Let x be a fractional assignment of restricted assignment jobs that is received online and let T be the fractional makespan of x, i.e. $T := \max_i \sum_{j \in N(i)} p_j x_{ij}$. There exists a randomized online algorithm that rounds a fractional assignment to an integer assignment such that the resulting makespan is at most $O((\log\log m)^3 T)$ with high probability.*

Proof. The result mostly follows from the lemmas in Section 3.5. The worst case for our algorithm is due to the large jobs with large support. In our algorithm there are $O(\log\log m)$ classes of large jobs by Lemma 3.5.2. Within a fixed class of large jobs there are $O(\log\log m)$ classes of large support jobs by Lemma 3.5.3. Finally, rounding a fixed class of large support jobs loses can be done with makespan $O((\log\log m)T)$ due to Lemma 3.5.10. The other cases of our algorithm lose fewer factors of $\log\log m$, and all cases of our algorithm succeed with high probability. Combining these losses we see that the makespan is at most $O((\log\log m)^3 T)$ with high probability. □

3.9 Removing Knowledge of T

Throughout this chapter, we described our algorithms and results as if we knew the optimal makespan T. We now show how to remove this assumption.

We use the following claim which states that we can combine k fractional solutions into a single fractional solution online while remaining $O(\log k)$-competitive against the best single fractional solution.

Claim 3.9.1. *Suppose that in addition to receiving each job j online, we also receive k fractional solutions $\{x_{ij}^s\}_{i\in[m],s\in[k]}$ which suggest how to assign job j. There is an online algorithm with fractional makespan at most $O(\log k)\cdot(\min_{s\in[k]}\max_{i\in[m]} p_{ij}x_{ij}^s)$*

We defer proving this claim here, but note that it is similar to results that have appeared before in the literature, e.g. that of Azar et al. [19].

Lemma 3.9.2. *There exists a fractional online algorithm using weight predictions which is $O(\log\log\log(m)\log(\eta))$-competitive which does not need to know the value of the optimal makespan T.*

Proof. The main idea is as follows. We apply the algorithm from Theorem 3.3.3 with $k = O(\log\log m)$ different guesses of T at once in parallel. We do this because we do not know the exact value of the error η, and we do not know the correct makespan T. This will produce k different fractional solutions, where ideally at least one of which will be $O(\log\eta)$-competitive. Then we can apply Claim 3.9.1 to conclude the lemma.

For each $s\in[k]$, where $k=\Theta(\log\log m)$, let T_s be the guess of T in the s'th parallel copy of our fractional algorithm which knows T i.e. copy s runs the algorithm from Theorem 3.3.3 with value T_s. We will always maintain that $T_{s+1}=2T_s$ for $s=2,3,\ldots,k-1$. We will start with $T_1=1$, and double the value of T_1 whenever the optimal makespan of the jobs seen so far increases by a constant factor. Updating T_1 causes us to update the other values of T_s by the logic above. Additionally, whenever we update the T_s values we reset each algorithm and simulate it on the already seen jobs with its new value of T_s. This incurs only a constant factor loss to the makespan of each copy. As discussed above, we just need to show that there always exists some parallel copy of our algorithm which is $O(\log\eta)$-competitive in order to conclude the lemma.

There are two cases. In the first case we have that $T_s \le T/\log(m)$ for some $s\in[k]$. In this case, we use the fact that the fractional algorithm from Theorem 3.3.3 will not be worse than $O(\log m)T_s = O(T) = O(\log(\eta)T)$. In the other case, we have that for all $s\in[k]$, $T/\log(m) < T_s \le T$. Since we spread the T_s values out in powers of 2, and there are $\Theta(\log\log(m)) = \Theta(k)$ powers of 2 in this interval, we conclude that some $T_s=\Theta(T)$. Thus by Theorem 3.3.3, there is a parallel copy of our algorithm with makespan $O(\log(\eta)T)$. □

Lemma 3.9.3. *There exists an online rounding algorithm using predictions yielding the same competitiveness as the algorithm guaranteed by Theorem 3.5.1 that does not need to know the optimal makespan T^* and succeeds with high probability.*

Proof. We consider running our algorithm with varying guesses of T. A **run** of the algorithm consists of starting with zero load on all machines and then assigning jobs while the makespan in this run is at most cT, where $c = O(\text{poly}(\log\log m))$ is the competitive ratio guaranteed by our algorithm. The run ends if an assignment would cause the makespan to go above cT. This event can be caused by at most three other events, (1) the random assignment of our algorithm failed with low probability, (2) the guess of T was wrong, or (3) the learned weights were wrong. Since bad random assignments happen with very low probability, we assume that this is not the case and in the end we union bound over all runs to conclude that we succeed with high probability. Thus we need to decide if the failure was caused by a wrong guess of T or bad weights.

To decide this question, we recompute the near optimal weights using Algorithm 9 with T as a guess of the makespan on the input received so far. If Algorithm 9 terminates with weights such that the resulting makespan is $\Omega(T)$, then the guess of the makespan was wrong and we start a new run with $T \leftarrow 2T$. Otherwise we start a new run with the same guess of T, and we use updated predictions of the weights.

The initial run starts with $T = 1$. Let T_f be the final guess of the makespan. Since our algorithm succeeds with high probability once $T \geq T^*$, we have that $T_f \leq 2T^*$. Our algorithm pays at most $2cT$ for each guess of T, since there might be one extra run where the predictions were bad. Let g be the number of guesses of T. Thus in total our algorithm incurs makespan at most

$$\begin{aligned}
&2cT_f + cT_f + cT_f/2 + cT_f/4 + \ldots + 2c \\
&= 2cT_f(1 + 1/2 + 1/4 + \ldots 1/2^g) \\
&\leq 2cT^*(1 + 1/2 + 1/4 + \ldots) \\
&= O(cT^*)
\end{aligned}$$

Say that a run is bad if the random assignment given by our rounding algorithm fails. The probability that a run is bad is at most $1/\operatorname{poly}(m)$. The number of runs is $O(\log(T^*)) = O(\operatorname{poly}(m))$, thus union bounding over all runs, we conclude that no run is bad with high probability, and thus the makespan bound above is correct with high probability. $\square$

Chapter 4

Speeding up the Hungarian Algorithm with Learned Duals

This chapter is based on "Faster Matchings via Learned Duals" [57], which appeared in the proceedings of Neural Information Processing Systems 2021 as an oral presentation. Collaborators on the project were Michael Dinitz, Sungjim Im, Benjamin Moseley, and Sergei Vassilvitskii.

4.1 Introduction

Classical algorithm analysis considers worst case performance of algorithms, capturing running times, approximation and competitive ratios, space complexities, and other notions of performance. Recently there has been a renewed interest in finding formal ways to go beyond worst case analysis [129], to better understand performance of algorithms observed in practice, and develop new methods tailored to typical inputs observed.

An emerging line of research dovetails this with progress in machine learning, and asks how algorithms can be augmented with machine-learned predictors to circumvent worst case lower bounds when the predictions are good, and approximately match them otherwise (see [116] for a survey). Naturally, a rich area of applications of this paradigm has been in online algorithms, where the additional information revealed by the predictions reduces the uncertainty about the future and can lead to better choices, and thus better competitive ratios. For instance, see the work by [108, 128, 85] on caching; [15, 62] on the classic secretary problem; [126, 100] on scheduling; [126, 11] on ski rental; and [31] on set cover.

However, the power of predictions is not limited to improving online algorithms. Indeed, the aim of the empirical paper that jump-started this area by [94] was to improve running times for basic indexing problems. The main goal and contribution of this work is to show that at least in one important setting (weighted bipartite matching), we can give formal justification for using machine learned predictions to improve running times: there are predictions which can provably be learned, and if these predictions are "good" then we have running times that outperform standard methods both in theory and empirically.

How can predictions help with running time? One intuitive approach, which has been used extensively in practice, is through the use of “warm-start” heuristics [151, 70, 71, 119], where instead of starting with a blank slate, the algorithm begins with some starting state (which we call a warm-start “solution” or “seed”) which hopefully allows for faster completion. While it is a common technique, there is a dearth of analysis understanding what constitutes a good warm-start, when such an initialization is helpful, and how they can best be leveraged.

Thus we have a natural goal: put warm-start heuristics on firm theoretical footing by interpreting the warm-start solution as learned predictions. In this set up we are given a number of instances of the problem (the training set), and we can use them to compute a warm-start solution that will (hopefully) allow us to more quickly compute the optimal solution on future, test-time, instances. There are three challenges that we must address:

(i) **Feasibility.** The learned prediction (warm-start solution) might not even be *feasible* for the specific instance we care about! For example, the learned solution may be matching an edge that does not exist in the graph at testing time.

(ii) **Optimization.** If the warm-start solution is feasible and near-optimal then we want the algorithm to take advantage of it. In other words, we would like our running time to be a function of the quality of the learned solution.

(iii) **Learnability.** It is easy to design predictions that are enormously helpful but which cannot actually be learned (e.g., the “prediction” is the optimal solution). We need to ensure that a typical solution learned from a few instances of the problem generalizes well to new examples, and thus offers potential speedups.

If we can overcome these three challenges, we will have an *end-to-end* framework for speeding up algorithms via learned predictions: use the solution to challenge (iii) to learn the predictions from historical data, use the solution to challenge (i) to quickly turn the prediction into something feasible for the particular problem instance while preserving near-optimality, and then use this as a warm-start seed in the solution to challenge (ii).

4.1.1 Our Contributions

We focus on one of the fundamental primitives of combinatorial optimization: computing bipartite matchings. For the bipartite minimum-weight perfect matching (MWPM) problem, as well as its extension to b-matching, we show that the above three challenges can be solved.

A key conceptual question is finding a specification of the seed, and an algorithm to use it that satisfies the desiderata above. We have discussed warm-start “solutions”, so it is tempting to think that a good seed is a partial solution: a set of matched edges that can then be expanded to a optimal matching. After all, this is the structure we maintain in most classical matching algorithms. Moreover, any such solution is feasible (one can simply set non-existing edges to have very high weight), eschewing the need for the feasibility step. At the same time, as has been observed previously in the context of online matchings [54, 145], this *primal* solution is brittle, and a minor modification in the instance (e.g. an addition of a single edge) can completely change the set of optimal edges.

Instead, following the work of [54, 145], we look at the *dual* problem; that is, the dual to the natural linear program. We quantify the "quality" of a prediction $\hat{y}$ by its ℓ_1-distance from the true optimal dual y^*, i.e., by $\|\hat{y} - y^*\|_1$. The smaller quantities correspond to better predictions. Since the dual is a packing problem we must contend with feasibility: we give a simple linear time algorithm that converts the prediction $\hat{y}$ into a feasible dual while increasing the ℓ_1 distance by a factor of at most 3.

Next, we run the Hungarian method starting with the resulting feasible dual. Here, we show that the running time is in proportional to the ℓ_1 distance of the feasible dual to the optimal dual (Theorem 4.3.11). Finally, we show via a pseudo-dimension argument that not many samples are needed before the empirically optimal seed is a good approximation of the true optimum (Theorem 4.3.12), and that this empirical optimum can be computed efficiently (Theorem 4.3.21). For the learning argument, we assume that matching instances are drawn from a fixed but unknown distribution $\mathcal{D}$.

Putting it all together gives us our main result.

Theorem 4.1.1 (Informal)**.** *There are three algorithms (feasibility, optimization, learning) with the following guarantees.*

- *Given a (possibly infeasible) dual $\hat{y}$ from the learning algorithm, there exists an $O(m + n)$ time algorithm that takes a problem instance c, and outputs a feasible dual $\hat{y}'(c)$ such that $\|\hat{y}'(c) - y^*(c)\|_1 \leq 3\|\hat{y} - y^*(c)\|_1$.*
- *The optimization algorithm takes as input feasible dual $\hat{y}'(c)$ and outputs a minimum weight perfect matching, and runs in time $\tilde{O}(m\sqrt{n} \cdot \min\{\|\hat{y}'(c) - y^*(c)\|_1, \sqrt{n}\})$.*
- *After $\tilde{O}(C^2 n^3)$ samples from an unknown distribution $\mathcal{D}$ over problem instances, the learning algorithm produces duals $\hat{y}$ so that $\mathbb{E}_{c\sim\mathcal{D}}\,[\|\hat{y} - y^*(c)\|_1]$ is approximately minimum among all possible choices of $\hat{y}$, where C is the maximum edge cost and $y^*(c)$ is an optimal dual for instance c.*

Combining these gives a single algorithm that, with access to $\tilde{O}(C^2 n^3)$ problem instance samples from $\mathcal{D}$, has expected running time on future instances from $\mathcal{D}$ of only $\tilde{O}(m\sqrt{n}\min\{\alpha, \sqrt{n}\})$, where $\alpha = \min_y \mathbb{E}_{c\sim\mathcal{D}}\,[\|y - y^*(c)\|_1]$.

We emphasize that the Hungarian method with $\tilde{O}(mn)$ running time is the standard algorithm in practice. Although there are other theoretically faster exact algorithms for bipartite minimum-weight perfect matching, such as those due to [123, 68, 66] and [60] that run in time $O(m\sqrt{n}\log(nC))$, they are relatively complex (using various scaling techniques). Very recent breakthroughs based on interior point methods give algorithms of run time $\tilde{O}((m + n^{1.5})\log^2(C))$ for the minimum-weight perfect matching problem and several interesting extensions [144, 143]. However, these breakthrough algorithms are highly complicated and their practical performance is yet to be demonstrated.

Note that our result shows that we can speed up the Hungarian method as long as the ℓ_1-norm error of the learned dual, i.e., $\|\hat{y} - y^*(c)\|_1$ is $o(\sqrt{n})$. Further, as the projection step that converts the learned dual into a feasible dual takes only linear time, the overhead of our method is essentially negligible. Therefore, even if the prediction is of poor quality, our method has worst-case running time that is *never* worse than that of the Hungarian algorithm. Even our learning algorithm is simple, consisting of a straightforward empirical risk minimization algorithm (the analysis is more complex and involves bounding the "pseudo-dimension" of the loss functions).

We validate our theoretical results via experiments. For each dataset we first feed a small number of samples (fewer than our theoretical bounds) to our learning algorithm. We then compare the running time of our algorithm to that of the classical Hungarian algorithm on new instances.

Details of these experiments can be found in Section 4.4. At a high level they show that our algorithm is *significantly* faster in practice. Further, our experiment shows only very few samples are needed to achieve a notable speed-up. This confirms the power of our approach, giving a theoretically rigorous yet also practical method for warm-start primal-dual algorithms.

4.1.2 Related Work

Matchings and b-Matchings: Bipartite matchings are one of the most well studied problems in combinatorial optimization, with a long history of algorithmic improvements. We refer the interested reader to [59] for an overview. We highlight some particular results here. If edges have no weights and thus the goal is to find the maximum cardinality matching (see Section 4.2 for a formal definition), the fastest running time had long been $O(m\sqrt{n})$ [56, 91, 79] until the recent breakthrough with $\tilde{O}(m + n^{1.5})$ running time was discovered [144]. We are interested in the weighted versions of these problems and when all edge weights are integral. Let C be the maximum edge weight, n be the number of vertices, and m the number of edges. For finding exact solutions to the minimum weight perfect matching problem, the scaling technique leads to a running time of $O(m\sqrt{n}\log(C))$ [123, 68, 66, 60].

The minimum cost b-matching problem and its generalization, the minimum cost flow problem, have also been extensively studied. See [121] for a summary of classical results. More recently there has been improvements by applying interior point methods. The algorithm of [52] has running time $\tilde{O}(m^{3/2}\log^2(C))$ and it is improved by the algorithm of [104] which runs in time $\tilde{O}(m\sqrt{n}\log^{O(1)}(C))$. We note that the recent breakthrough of van den Brand et al. [144] solves the minimum cost flow problem in time $\tilde{O}((m + n^{1.5})n^{o(1)}\log^2(BC))$, where B is the maximum capacity and C is the maximum edge weight.

Large scale bipartite matchings have been studied extensively in the online setting, as they represent the basic problem in ad allocations [111]. While the ad allocation is inherently online, most of the methods precompute a dual based solution based on a sample of the input [54, 145], and then argue that this solution is approximately optimal on the full instance. In contrast, we strive to compute the *exactly optimal* solution, but use previous instances to improve the running time of the approach.

We refer the reader back to the introduction of this dissertation for an overview of recent work on algorithms with predictions and data-driven algorithm design.

4.1.3 Roadmap

We begin with preliminaries and background in Section 4.2. We then present our main theoretical results on min-cost perfect bipartite matching in Section 4.3. The experiments are presented in Section 4.4. Finally, the extension to b-matching is presented in Section 4.5.

4.2 Preliminaries

Notation: Let $G = (V, E)$ be an undirected graph. When G is bipartite we will use L and R to refer to the two sides of the bipartition. We will let $N(i) := \{e \in E \mid i \in e\}$ be the set of edges adjacent to vertex i. Similarly if G is directed, then we use $N^+(i)$ and $N^-(i)$ to be the set of edges leaving i and the set of edges entering i, respectively. For a set $S \subseteq V$, let $\Gamma(S)$ be the vertex neighborhood of S. For a vector $y \in \mathbb{R}^n$, we let $\|y\|_1 = \sum_i |y_i|$ be its ℓ_1-norm. Let $\langle x, y\rangle$ be the standard inner product on $\mathbb{R}^n$.

Linear Programming and Complementary Slackness: Here we recall optimality conditions for linear programming that are used to ensure the correctness of some algorithms we present. Consider the primal-dual pair of linear programs below.

$$\begin{aligned} \min \quad & c^\top x \\ & Ax = b \\ & x \geq 0 \end{aligned} \tag{P}$$

$$\begin{aligned} \max \quad & b^\top y \\ & A^\top y \leq c \end{aligned} \tag{D}$$

A pair of solutions x, y for (P) and (D), respectively, satisfy complementary slackness if $x^\top(c - A^\top y) = 0$. The following lemma is well-known.

Lemma 4.2.1. *Let x be a feasible solution for (P) and y be a feasible solution for (D). If the pair x, y satisfies complementary slackness, then x and y are optimal solutions for their respective problems.*

Maximum Cardinality Matching: Let $G = (V, E)$ be a bipartite graph on n vertices and m edges. A matching $M \subseteq E$ is a collection of non-intersecting edges. The Hopcroft-Karp algorithm for finding a matching maximizing $|M|$ runs in time $O(\sqrt{n} \cdot m)$ [79], which is still state-of-the-art for general bipartite graphs. For moderately dense graphs, a recent result by [144] gives a better running time of $\tilde{O}(m + n^{1.5})$ (where $\tilde{O}$ hides polylogarithmic factors).

Minimum Weight Perfect Matching (MWPM): Again, let $G = (V, E)$ be a bipartite graph on n vertices and m with costs $c \in \mathbb{Z}_+^E$ on the edges, and let C be the maximum cost. A matching M is *perfect* if every vertex is matched by M. The objective of this problem is to find a perfect matching M minimizing the cost $c(M) := \sum_{e \in M} c_e$.

When looking for optimal solutions we can assume that G is a complete graph by adding all possible edges not in E with weight Cn^2. It is easy to see that any $o(n)$ approximate solution would not use any of these edges.

Maximum Flow: Now let $G = (V, E)$ be a directed graph on n vertices and m edges with a capacity vector $u \in \mathbb{R}^E_+$. Let s and t be distinct vertices of G. An st-flow is a vector $f \in \mathbb{R}^E_+$ satisfying $\sum_{e \in N^+(i)} f_e - \sum_{e \in N^-(i)} f_e = 0$ for all vertices $i \neq s, t$. An st-flow f is maximum if it maximizes $\sum_{e \in N^+(s)} f_e = \sum_{e \in N^-(t)} f_e$. The algorithm due to Orlin [122] and King, Rao, and Tarjan [92] runs in time $O(nm)$.

4.3 Faster Min-Weight Perfect Matching

In this section we describe how predictions can be used to speed up the bipartite Minimum Weight Perfect Matching (MWPM) problem.

The MWPM problem can be modeled by the following linear program and its dual – the primal-dual view will be very useful for our algorithm and analysis. We will sometimes refer to a set of dual variables y as dual *prices*. Both LPs are well-known to be integral, implying that there always exist integral optimal solutions.

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ & \sum_{e \in N(i)} x_e = 1 \quad \forall i \in V \\ & x_e \geq 0 \qquad \forall e \in E \end{aligned} \tag{MWPM-P}$$

$$\begin{aligned} \max \quad & \sum_{i \in V} y_i \\ & y_i + y_j \leq c_e \quad \forall e = ij \in E \end{aligned} \tag{MWPM-D}$$

Suppose we are given a prediction $\hat{y}$ of a dual solution. If $\hat{y}$ is feasible, then by complementary slackness we can check if $\hat{y}$ represents an optimal dual solution by running a maximum cardinality matching algorithm on the graph $G' = (V, E')$, where $E' = \{e = ij \in E \mid \hat{y}_i + \hat{y}_j = c_{ij}\}$ is the set of tight edges. If this matching is perfect, then its incidence vector x satisfies complementary slackness with $\hat{y}$ and thus represents an optimal solution by Lemma 4.2.1.

We now consider the problem from another angle, factoring in learning aspects. Suppose the graph $G = (V, E)$ is fixed but the edge cost vector $c \in \mathbb{Z}^E_+$ varies (is drawn from some distribution $\mathcal{D}$). If we are given an optimal dual y^* as a prediction, then we can solve the problem by solving the max cardinality matching problem only once. However, the optimal dual can significantly change depending on edge cost c. Nevertheless, we will show how to learn "good" dual values and use them later to solve new MWPM instances faster. Specifically, we seek to design an end-to-end algorithm addressing all the aforementioned challenges:

1. **Feasiblity** (Section 4.3.1). The learned dual $\hat{y}$ may not be feasible for MWPM-D with some specific cost vector c. We show how to quickly convert it to a feasible dual $\hat{y}'(c)$ by appropriately decreasing the dual values (the more we decrease them, the further we move away from the optimum). Finding the feasible dual minimizing $\|\hat{y} - \hat{y}'(c)\|_1$ turns out to be a variant of the vertex cover problem, for which we give a simple 2-approximation running in $O(m + n)$ time. As a result, we have $\|\hat{y}'(c) - y^*(c)\|_1 \leq 3\|\hat{y} - y^*(c)\|_1$. See Theorem 4.3.5.

2. **Optimization** (Section 4.3.2). Now that we have a feasible solution $\hat{y}'(c)$, we want to find an optimal solution starting with $\hat{y}'(c)$ in time that depends on the quality of $\hat{y}'(c)$. Fortunately, the Hungarian algorithm can be seeded with any feasible dual, so we can "warm-start" it with $\hat{y}'(c)$. We show that its running time will be proportional to $|\|\hat{y}'(c)\|_1 - \|y^*(c)\|_1| \leq \|\hat{y}'(c) - y^*(c)\|_1$.See Theorem 4.3.11. Our analysis does not depend on the details of the Hungarian algorithm, and so applies to a broader class of primal-dual algorithms.

3. **Learnability** (Section 4.3.3). The target dual we seek to learn is given by $\arg\min_y \mathbb{E}_{c\sim\mathcal{D}}\|y - y^*(c)\|$; here $y^*(c)$ is the optimal dual for MWPM-D with cost vector c. We show we can efficiently learn $\hat{y}$ that is arbitrarily close to the target vector after $\tilde{O}(C^2n^3)$ samples from $\mathcal{D}$. See Theorem 4.3.12.

Combining all of these gives the following, which is a more formal version of Theorem 4.1.1. Let $\mathcal{D}$ be an arbitrary distribution over edge costs where every vector in the support of $\mathcal{D}$ has maximum cost C. For any edge cost vector c, let $y^*(c)$ denote the optimal dual solution.

Theorem 4.3.1. *For any $p, \epsilon > 0$, there is an algorithm which:*

- *After $O\left(\left(\frac{nC}{\epsilon}\right)^2 (n\log n + \log(1/p))\right)$ samples from $\mathcal{D}$, returns dual values $\hat{y}$ such that $\mathbb{E}_{c\sim\mathcal{D}}[\|\hat{y} - y^*(c)\|_1] \leq \min_y \mathbb{E}_{c\sim\mathcal{D}}[\|y - y^*(c)\|_1] + \epsilon$ with probability at least $1 - p$.*

- *Using the learned dual $\hat{y}$, given edge costs c, computes a min-cost perfect matching in time $O\left(m\sqrt{n} \cdot \min\{\|\hat{y} - y^*(c)\|_1, \sqrt{n}\}\right)$.*

In the rest of this section we detail our proof of this Theorem.

4.3.1 Recovering a Feasible Dual Solution (Feasibility)

Let $\hat{y}$ be an infeasible set of (integral) dual prices – this should be thought of as the "good" dual obtained by our learning algorithm. Our goal in this section is to find a new *feasible* dual solution $\hat{y}'(c)$ that is close to $\hat{y}$, for a given MWPM-D instance with cost c. In particular we seek to find the closest feasible dual under the ℓ_1 norm, i.e. one minimizing $\|\hat{y}'(c) - \hat{y}\|_1$.

Looking at (MWPM-D), it is clear that we need to decrease the given dual values $\hat{y}$ in order to make it feasible. More formally, we are looking for a vector of non-negative perturbations δ such that $\hat{y}' := \hat{y} - \delta$ is feasible. We model finding the best set of perturbations, in terms of preserving $\hat{y}$'s dual objective value, as a linear program. Let $F := \{e = ij \in E \mid \hat{y}_i + \hat{y}_j > c_{ij}\}$ be the set of dual infeasible edges under $\hat{y}$. Define $r_e := \hat{y}_i + \hat{y}_j - c_e$ for each edge $e = ij \in F$. Asserting that $\hat{y} - \delta$ is feasible for (MWPM-D) while minimizing the amount lost in the dual objective leads to the following linear program:

$$\begin{aligned} \min \quad & \sum_{i \in V} \delta_i \\ & \delta_i + \delta_j \geq r_{ij} \quad \forall ij \in F \\ & \delta_i \geq 0 \qquad \forall i \in V \end{aligned} \tag{4.1}$$

Note that this is a variant of the vertex cover problem—the problem becomes exactly the vertex cover problem if $r_{ij} = 1$ for all edges ij. We could directly solve this linear program, but we are interested in making this step efficient. To find a fast approximation for (4.1), we take a simple greedy approach.

Algorithm 10 Fast Approx. for Distance to Feasibility

```
 1: procedure FastApprox($G = (V, E), r$)
 2:     $\forall i \in V$, $\delta_i \leftarrow 0$
 3:     while $E \neq \emptyset$ do
 4:         Let $i$ be an arbitrary vertex of $G$
 5:         while $i$ has a neighbor do
 6:             $j \leftarrow \arg\max_{j' \in N(i)} r_{ij'}$
 7:             $\delta_i \leftarrow r_{ij}$
 8:             $\gamma_{ij} = 1/2$                    ▷ $\gamma_{ij}$ is only used for analysis
 9:             Delete $i$ and all its edges from $G$
10:             $i \leftarrow j$
11:         end while
12:     end while
13:     Return $\delta$
14: end procedure
```

Algorithm 10 is a modification of the algorithm of [58] which walks through the graph setting δ_i appropriately at each step to satisfy the covering constraints in (4.1). The analysis is based on interpreting the algorithm through the lens of primal-dual—the dual of (4.1) turns out to be a maximum weight matching problem with new edge weights r_{ij}.

The dual is the following:

$$\begin{aligned} \max \quad & \sum_{e} r_e \gamma_e \\ & \sum_{e \in N(i) \cap F} \gamma_e \leq 1 \quad \forall i \in V \\ & \gamma_e \geq 0 \qquad \forall e \in F \end{aligned} \tag{4.2}$$

First we show the algorithm is fast.

Lemma 4.3.2. *Algorithm 10 runs in time $O(n+m)$.*

Proof. This follows from the trivial observation that each vertex/edge is considered $O(1)$ times. $\square$

Next, we show that the algorithm constructs a feasible dual solution.

Lemma 4.3.3. *The perturbations δ returned by Algorithm 10 is feasible for* (4.1)*.*

Proof. We want to show that $\delta_i + \delta_j \geq r_e$ for all edges $e = ij \in E$. We claim that this condition holds whenever the edge is deleted from G. Suppose that the algorithm is currently at i and let ij' be the edge selected by the algorithm in this step. By definition of the algorithm we have $\delta_i = r_{ij'} \geq r_{ij}$ so $\delta_i + \delta_j \geq r_{ij}$. $\square$

Finally, we address the objective.

Lemma 4.3.4. *The perturbations δ returned by Algorithm 10 are a 2-approximation for* (4.1)*.*

Proof. In each iteration, the increase of the primal objective ($\delta_i = r_{ij}$ in Line 7) is exactly twice the increase of the dual objective ($r_{ij}\gamma_{ij} = r_{ij}/2$ in Line 8). Thus, due to weak duality, it suffices to show that the dual is feasible. This follows from the observation that $\{ij \mid \gamma_{ij} = 1/2\}$ forms a collection of vertex disjoint paths and cycles. Thus, for every $i \in V$, there are at most two edges e adjacent to i such that $\gamma_e > 0$, and for those edges e, $\gamma_e = 1/2$. Therefore, the dual is feasible. $\square$

This shows we can project the predicted dual prices $\hat{y}$ onto the set of feasible dual prices at approximately the minimum cost such way of doing so. The prior lemmas give the following theorem by noticing that $y^*(c)$ is a possible feasible solution. Note that integrality is immediate from the algorithm.

Theorem 4.3.5. *There is a $O(m+n)$ time algorithm that takes an infeasible integer dual $\hat{y}$ and constructs a feasible integer dual $\hat{y}'(c)$ for MWPM-D with cost vector c such that $\|\hat{y}'(c) - \hat{y}\|_1 \leq 2\|\hat{y} - y^*(c)\|_1$ where $y^*(c)$ is the optimal dual solution for MWPM-D with cost vector c. Thus by triangle inequality we have $\|\hat{y}'(c) - y^*(c)\|_1 \leq 3\|\hat{y} - y^*(c)\|_1$.*

4.3.2 Seeding Hungarian with a Feasible Dual (Optimization)

In this section we assume that we are given a feasible integral dual $\hat{y}'(c)$ for an input with cost vector c and the goal is to find an optimal solution. We want to analyze the running time in terms of $\|\hat{y}'(c) - y^*(c)\|_1$, the distance to optimality. We use a simple primal-dual schema to achieve this, which is given formally in Algorithm 11.

Algorithm 11 Simple Primal-Dual Scheme for MWPM

```
1: procedure MWPM-PD($G = (V, E), c, y$)
2:     $E' \leftarrow \{e \in E \mid y_i + y_j = c_{ij}\}$        ▷ Set of tight edges in the dual
3:     $G' \leftarrow (V, E')$        ▷ $G$ containing only tight edges
4:     $M \leftarrow$ Maximum cardinality matching in $G'$
5:     while $M$ is not a perfect matching do
6:         Find $S \subseteq L$ such that $|S| > |\Gamma(S)|$ in $G'$        ▷ Exists by Hall's Theorem
                                                  ▷ Can be found in $O(m + n)$ time
7:         $\epsilon \leftarrow \min_{i \in S, j \in R \setminus \Gamma(S)}\{c_{ij} - y_i - y_j\}$
8:         $\forall i \in S,\ y_i \leftarrow y_i + \epsilon$
9:         $\forall j \in \Gamma(S),\ y_j \leftarrow y_j - \epsilon$
10:        Update $E', G'$
11:        $M \leftarrow$ Maximum cardinality matching in $G'$
12:    end while
13:    Return $M$
14: end procedure
```

To satisfy complementary slackness, we must only choose edges with $y_i + y_j = c_{ij}$. Let E' be the set of such edges. We find a maximum cardinality matching in the graph $G' = (V, E')$. If the resulting matching M is perfect then we are done by complementary slackness (Lemma 4.2.1) Otherwise, in steps 7-9 we modify the dual in a way that guarantees a strict increase in the dual objective. Since all parameters of the problem are integral, this strict increase then implies our desired bound on the number of iterations.

We now analyze Algorithm 11. Recall that L and R give the bipartition of V. First we show that the algorithm is correct. The main claim we need to establish is that if y is initially dual feasible, then it remains dual feasible throughout the algorithm. First we check that the update defined in lines 6-10 is well defined, i.e. in line 6 such a set S always exists and ϵ defined in line 7 is always strictly positive.

Proposition 4.3.6. *If M is not a perfect matching in G', then there exists a set $S \subseteq L$ such that $|S| > |\Gamma(S)|$ in G'. Further, such S can be found in $O(m+n)$ time.*

Proof. The first claim follows directly from Hall's Theorem applied to G'. It is well-known that the maximum matching size is equal to the minimum vertex cover size when the underlying graph is bipartite. Further, a minimum vertex cover C can be derived from a maximum matching M in time $O(m+n)$. We set $S = L \setminus C$. Then, we have $\Gamma(S) \subseteq C \cup R$ due to C being a vertex cover, and $|C \cap L| + |C \cap R| = |C| < n$ as the minimum cover size is less than n; recall M is not perfect. Thus, we have $|S| = n - |C \cap L| > |C \cap R| \geq |\Gamma(S)|$, as desired. □

Proposition 4.3.7. *Let y be dual feasible and suppose that $S \subseteq L$ with $|S| > |\Gamma(S)|$ in G'. Let $\epsilon = \min_{i \in S, j \in R \setminus \Gamma(S)} c_{ij} - y_i - y_j$. Then as long as c and y are integer we have $\epsilon \geq 1$.*

Proof. Every edge ij considered in the definition of ϵ is not in E' and thus must have $c_{ij} > y_i + y_j$. Thus for all such edges we have $c_{ij} - y_i - y_j \geq 1$ since c and y are integer, and so $\epsilon \geq 1$.

If no such edge exists, then we have a set $S \subseteq L$ such that $|S|$ is strictly larger than its neighborhood in G (rather than G') which shows that the problem is infeasible. This contradicts our assumption that the original problem is feasible. $\square$

We now show the main claims we described above.

Lemma 4.3.8. *If Algorithm 11 is given an initial dual feasible y, then y remains dual feasible throughout its execution.*

Proof. Inductively, it suffices to show that if y is dual feasible then it remains so after the update steps defined in lines 6-10. To make the notation clear, let y' be the result of applying the update rule to y. Consider an edge $ij \in E$. We want to show that $y'_i + y'_j \leq c_{ij}$ after the update step. There are 4 cases to check: (1) $i \in L \setminus S, j \in R \setminus \Gamma(S)$, (2) $i \in L \setminus S, j \in \Gamma(S)$, (3) $i \in S, j \in R \setminus \Gamma(S)$, and (4) $i \in S, j \in \Gamma(S)$.

In the first case, neither y_i nor y_j are modified, so we get $y'_i + y'_j = y_i + y_j \leq c_{ij}$ since y was initially dual feasible. In the second case we have $y'_i + y'_j = y_i + y_j - \epsilon \leq c_{ij}$ since $\epsilon > 0$. In the third case we have $y'_i = y_i + \epsilon$ and so $y'_i + y'_j = y_i + \epsilon + y_j \leq c_{ij}$ since there was slack on these edges and ϵ was chosen to be the smallest such slack. Finally, in the last case we have $y'_i + y'_j = y_i + \epsilon + y_j - \epsilon \leq c_{ij}$. Thus we conclude that y remains feasible throughout the execution of Algorithm 11. $\square$

Lemma 4.3.9. *Each iteration strictly increases the value of the dual solution.*

Proof. Note that in each iteration y_i increases by ϵ for all $i \in S$ and y_j decreases by ϵ for all $j \in \Gamma(S)$; and all other dual variables remain unchanged. Thus, the dual objective increases by $\epsilon(|S| - |\Gamma(S)|) \geq \epsilon$. $\square$

The above lemma allows us to analyze the running time of our algorithm in terms of the distance to optimality.

Lemma 4.3.10. *Consider an arbitrary cost vector c. Suppose that $\hat{y}'(c)$ is an integer dual feasible solution and $y^*(c)$ is an integer optimal dual solution. If Algorithm 11 is initialized with $\hat{y}'(c)$, then the number of iterations is bounded by $\|\hat{y}(c) - y^*(c)\|_1$.*

Proof. By Lemma 4.3.9, we have that the value of the dual solution increases by at least 1 in each iteration. Thus the number of iterations is at most $\sum_i y^*_i(c) - \sum_i \hat{y}_i(c) \leq \sum_i |y^*_i(c) - \hat{y}_i(c)| = \|y^*(c) - \hat{y}(c)\|_1$. $\square$

Finally, we get the following theorem as a corollary of the lemmas above and the $O(m\sqrt{n})$ runtime of the Hopcroft-Karp algorithm for maximum cardinality matching [79]. More precisely, the above lemmas show that the algorithm performs at most $O\left(\|y^*(c) - \hat{y}'(c)\|_1\right)$ iterations, each running in $O(m\sqrt{n})$ time. We can further improve this by ensuring the algorithm runs no longer than the standard Hungarian algorithm in the case that we have large error in the prediction, i.e., $\|y^*(c) - \hat{y}'(c)\|_1$ is large. In particular, steps 6 and 11 do not precisely specify the choice of the set S and the matching M. If we instantiate these steps appropriately (let $S = L \setminus C$ for step 6, where C is a minimum vertex cover, and update M along shortest-augmenting-paths for step 11) then we recover the Hungarian Algorithm and its $\tilde{O}(mn)$ running time.

Theorem 4.3.11. *Consider an arbitrary cost vector c. There exists an algorithm which takes as input a feasible integer dual solution $\hat{y}'(c)$ and finds a minimum weight perfect matching in $\tilde{O}\left(\min\{m\sqrt{n}\|y^*(c) - \hat{y}'(c)\|_1, mn\}\right)$ time, where $y^*(c)$ is an optimal dual solution.*

4.3.3 Learning Optimal Advice (Learning)

Now we want to formally instantiate the "learning" part of our framework: if there is a good starting dual solution for a given input distribution, we want to find it without seeing too many samples. The formal model we will use is derived from data driven algorithm design and PAC learning.

We imagine solving many problem instances drawn from the same distribution. To formally model this, we let $\mathcal{D}$ be an unknown distribution over instances. For simplicity, we consider the graph $G = (V, E)$ to be fixed with varying costs. Thus $\mathcal{D}$ is a distribution over cost vectors $c \in \mathbb{R}^E$. We assume that the costs in this distribution are bounded. Let $C := \max_{c \sim \mathcal{D}} \max_{e \in E} c_e$ be finite and known to the algorithm. Our goal is to find the (not necessarily feasible) dual assignment that performs "best" in expectation over the distribution. Based on Theorems 4.3.5 and 4.3.11 , we know that the "cost" of using dual values y when the optimal dual is y^* is bounded by $O(m\sqrt{n}\|y^* - y\|_1)$, and hence it is natural to define the "cost" of y as $\|y^* - y\|_1$.

For every $c \in \mathbb{R}^E$ we will let $y^*(c)$ be a fixed optimal dual solution for c:

$$y^*(c) := \arg\max_y \Big\{ \sum_i y_i \mid \forall ij \in E, y_i + y_j \leq c_{ij} \Big\}.$$

Here we assume without loss of generality that $y^*(c)$ is integral as the underlying polytope is known to be integral. We will let the loss of a dual assignment y be its ℓ_1-distance from the optimal solution:

$$L(y, c) = \|y - y^*(c)\|_1.$$

Our goal is to learn dual values $\hat{y}$ which minimize $\mathbb{E}_{c \sim \mathcal{D}}[L(y, c)]$. Let y^* denote the vector minimizing this objective, $y^* = \arg\min_y \mathbb{E}_{c \sim \mathcal{D}}[L(y, c)]$.

We will give PAC-style bounds, showing that we only need a small number of samples in order to have a good probability of learning an approximately-optimal solution $\hat{y}$. Our algorithm is conceptually quite simple: we minimize the empirical loss after an appropriate number of samples. We have the following theorem.

Theorem 4.3.12. *There is an algorithm that after $s = O\left(\left(\frac{nC}{\epsilon}\right)^2 (n \log n + \log(1/p))\right)$ samples returns dual values $\hat{y}$ such that $\mathbb{E}_{c \sim \mathcal{D}}[L(\hat{y}, c)] \leq \mathbb{E}_{c \sim \mathcal{D}}[L(y^*, c)] + \epsilon$ with probability at least $1 - p$. The algorithm runs in time polynomial in n, m and s.*

This theorem, together with Theorems 4.3.5 and 4.3.11, immediately implies Theorem 4.3.1.

Proof of Theorem 4.3.12

We now discuss the main tools we require from statistical learning theory in order to prove Theorem 4.3.12. For every dual assignment $y \in \mathbb{R}^V$, we define a function $g_y : \mathbb{R}^E \to \mathbb{R}$ by $g_y(c) = L(y, c) = \|y^*(c) - y\|_1$. Let $\mathcal{H} = \{g_y \mid y \in \mathbb{R}^V\}$ be the collection of all such functions. It turns out that in order to prove Theorem 4.3.12, we just need to bound the *pseudo-dimension* of this collection. Note that the notion of shattering and pseudo-dimension in the following is a generalization to real-valued functions of the classical notion of VC-dimension for Boolean-valued functions (classifiers).

Definition 4.3.13 ([124, 13]). *Let $\mathcal{F}$ be a class of functions $f : X \to \mathbb{R}$. Let $S = \{x_1, x_2, \ldots, x_s\} \subset X$. We say that that S is* shattered *by $\mathcal{F}$ if there exist real numbers $r_1, \ldots, r_s$ so that for all $S' \subseteq S$, there is a function $f \in \mathcal{F}$ such that $f(x_i) \leq r_i \iff x_i \in S'$ for all $i \in [s]$. The* pseudo-dimension *of $\mathcal{F}$ is the largest s such that there exists an $S \subseteq X$ with $|S| = s$ that is shattered by $\mathcal{F}$.*

The connection between pseudo-dimension and learning is given by the following uniform convergence result.

Theorem 4.3.14 ([124, 13]). *Let $\mathcal{D}$ be a distribution over a domain X and $\mathcal{F}$ be a class of functions $f : X \to [0, H]$ with pseudo-dimension $d_{\mathcal{F}}$. Consider s independent samples $x_1, x_2, \ldots, x_s$ from $\mathcal{D}$. There is a universal constant c_0, such that for any $\epsilon > 0$ and $p \in (0, 1)$, if $s \geq c_0 \left(\frac{H}{\epsilon}\right)^2 (d_{\mathcal{F}} + \ln(1/p))$ then we have*

$$\left| \frac{1}{s} \sum_{i=1}^{s} f(x_i) - \mathbb{E}_{x \sim \mathcal{D}}[f(x)] \right| \leq \epsilon$$

for all $f \in \mathcal{F}$ with probability at least $1 - p$.

Intuitively, this theorem says that the sample average $\frac{1}{s}\sum_{i=1}^{s} f(x_i)$ is close to its expected value for every function $f \in \mathcal{F}$ simultaneously with high probability so long as the sample size s is large enough. This theorem can be utilized to give a learning algorithm for our problem by considering an algorithm which minimizes the empirical loss. In general, the "best" function is the one which minimizes the expected value over $\mathcal{D}$, i.e. $f^* = \arg\min_{f \in \mathcal{F}} \mathbb{E}_{x \sim \mathcal{D}}[f(x)]$. We have the following simple corollary for learning and approximately best function $\hat{h}$.

Corollary 4.3.15. *Consider a set of s independent samples $x_1, x_2, \ldots, x_s$ from $\mathcal{D}$ and let $\hat{f}$ be a function in $\mathcal{F}$ which minimizes $\frac{1}{s}\sum_{i=1}^{s} \hat{f}(x_i)$. If s is chosen as in Theorem 4.3.14, then with probability $1 - p$ we have $\mathbb{E}_{x \sim \mathcal{D}}[\hat{f}(x)] \leq \mathbb{E}_{x \sim \mathcal{D}}[f^*(x)] + 2\epsilon$*

Thus based on the above Theorem and Corollary, to prove Theorem 4.3.12 we must accomplish the following tasks. First and foremost, we must bound the pseudo-dimension of our class of functions $\mathcal{H}$. Next, we need to check that the functions are bounded on the domain we consider, and finally we need to give an algorithm minimizing the empirical risk. The latter two tasks are simple. By assumption the edge costs are bounded by C.

If we restrict $\mathcal{H}$ to be within a suitable bounding box, then one can verify that we can take $H = O(nC)$ to satisfy the conditions for Theorem 4.3.14. Additionally, the task of finding a function to minimize the loss on the sample can be done via linear programming. We formally verify these details in Sections 4.3.3 and 4.3.3. This leaves bounding the pseudo-dimension of the class $\mathcal{H}$, which we focus on now.

To bound the pseudo-dimension of $\mathcal{H}$, we will actually consider a different class of functions $\mathcal{H}_n$: for every $y \in \mathbb{R}^n$ we define a function $f_y : \mathbb{R}^n \to \mathbb{R}$ by $f_y(x) = \|y - x\|_1$, and we let $\mathcal{H}_n = \{f_y \mid y \in \mathbb{R}^n\}$. It is not hard to argue that it is sufficient to bound the pseudo-dimension of this class.

Lemma 4.3.16. *If the pseudo-dimension of $\mathcal{H}_n$ is at most k, then the pseudo-dimension of $\mathcal{H}$ is at most k.*

Proof. We prove the contrapositive: we start with a set of size s which is shattered by $\mathcal{H}$, and use it to find a set of size s which is shattered by $\mathcal{H}_n$. Let $S = \{c_1, c_2, \ldots, c_s\}$ with each $c_i \in \mathbb{R}^E$ be a set which is shattered by $\mathcal{H}$. Then there are real numbers $r_1, r_2, \ldots, r_s$ so that for all $S' \subseteq [s]$, there is a function $g \in \mathcal{H}$ where $g(c_i) \leq r_i \iff i \in S'$. By definition of $\mathcal{H}$, this g is $g_{y_{S'}}$ for some $y_{S'} \in \mathbb{R}^n$, and so $\|y_{S'} - y^*(c_i)\|_1 \leq r_i \iff i \in S'$.

Let $\hat{S} = \{y^*(c_1), y^*(c_2), \ldots, y^*(c_s)\}$. We claim that $\hat{S}$ is shattered by $\mathcal{H}_n$. To see this, consider the same real numbers $r_1, \ldots, r_s$ and some $S' \subseteq [s]$. Then $f_{y_{S'}}(y^*(c_i)) = \|y_{S'} - y^*(c_i)\|_1 = g_{y_{S'}}(c_i)$ and hence $f_{y_{S'}}(y^*(c_i)) \leq r_i \iff g_{y_{S'}}(c_i) \leq r_i \iff i \in S'$. Thus $\hat{S}$ is shattered by $\mathcal{H}_n$. $\square$

So now our goal is to prove the following bound, which (with Lemma 4.3.16 and Theorem 4.3.14) implies Theorem 4.3.12.

Theorem 4.3.17. *The pseudo-dimension of $\mathcal{H}_n$ is at most $O(n \log n)$.*

Let k be the pseudo-dimension of $\mathcal{H}_n$. Then by the definition of pseudo-dimension there is a set $P = \{x^1, x^2, \ldots, x^k\}$ which is shattered by $\mathcal{H}_n$, so there are values $r_1, r_2, \ldots, r_k \in \mathbb{R}_{\geq 0}$ so that for all $S \subseteq P$ there is an $f \in \mathcal{H}_n$ such that $f(x^i) \leq r_i \iff x^i \in S$. By our definition of $\mathcal{H}_n$, this means that there is a $y_S \in \mathbb{R}^n$ so that $\|y_S - x^i\|_1 \leq r_i \iff x^i \in S$.

For each $S \subseteq P$, define the *region* of S (denoted by $r(S)$) to be

$$r(S) = \{y \in \mathbb{R}^n : \|y - x^i\|_1 \leq r_i \iff x^i \in S\},$$

i.e., the set of points that are at ℓ_1-distance at most r_i from x^i for precisely the x^i's that are in S. Clearly each $r(S)$ is nonempty for every $S \subseteq P$ due to the existence of y_S. Let $m = 2^k$ be the number of nonempty regions.

To upper bound the pseudo-dimension k we will prove that there cannot be too many nonempty regions (i.e., m is small). This is somewhat complex since the ℓ_1-balls have complex structure (in particular, they have 2^n facets), so we will do this by partitioning $\mathbb{R}^n$ into *cells* in which the ℓ_1 balls are simpler. For each $x^i \in P$ and $j \in [n]$, let Q^i_j be the hyperplane in $\mathbb{R}^n$ that passes through y_i and is perpendicular to the axis e_j (i.e., $Q^i_j = \{y \in \mathbb{R}^n : \langle y - x^i, e_j \rangle = 0\}$). Clearly there are kn of these hyperplanes. Define a *cell* to be a maximal set of points in $\mathbb{R}^n$ which are the same side of every hyperplane. Note that there are $(k+1)^n$ of these cells, they partition $\mathbb{R}^n$, and every cell which is bounded is a hypercube.

Lemma 4.3.18. *Let C be a cell and $x^i \in P$. There is a halfspace H such that $B_1(x^i, r_i) \cap C = H \cap C$.*

Proof. If $B_1(x^i, r_i) \cap C = \emptyset$ then we are done. So suppose that $B_1(x^i, r_i) \cap C \neq \emptyset$. By definition, $B_1(x^i, r_i)$ is the set of points $y \in \mathbb{R}^n$ such that $\sum_{j=1}^n |x^i_j - y_j| \leq r_i$. Hence $B_1(x^i, r_i)$ is defined by the intersection of 2^n halfspaces:

$$B_1(x^i, r_i) = \left\{ y \in \mathbb{R}^n \ \mid \ \sum_{j=1}^n a_j(x^i_j - y_j) \leq r_i \ \forall a \in \{-1, +1\}^n \right\}$$

If the intersection of the boundary of $B_1(x^i, r_i)$ with C is one of these hyperplanes, then we are finished. Otherwise, there are at least two of these hyperplanes $H_1 = (a_1, \dots a_n)$ and $H_2 = (a'_1, \dots, a'_n)$ such that $B_1(x^i, r_i) \cap C$ contains a point $y \in H_1 \setminus H_2$ and a point $y' \in H_2 \setminus H_1$, both of which are also on the boundary of $B_1(x^i, r_i)$. Let $j \in [n]$ such that $a_j = -a'_j$. Then $y_j - x^i_j$ has a different sign than $y'_j - x^i_j$, since the fact that y and y' are on the boundary of $B_1(x^i, r_i)$ but on different facets implies that $x^i_j - y_j$ has sign a_j while $x^i_j - y'_j$ has sign a'_j. But this contradicts the definition of C, since it means that y and y' are on different sides of Q^i_j and hence not in the same cell. □

This lemma allows us to analyze the number of regions that intersect any cell.

Lemma 4.3.19. *Let C be a cell. The number of regions that intersect C is at most $2^{O(n)}k^n$.*

Proof. For every $S \subseteq P$, the region $r(S)$ is the set of points that are in $B_1(x^i, r_i)$ for all $x_i \in S$ and are not in $B_1(x^i, r_i)$ for all $x_i \notin S$. By Lemma 4.3.18, $r(S) \cap C$ is the intersection of C with k halfspaces (one for each $x^i \in P$). It is well-knownthat k halfspaces can divide $\mathbb{R}^n$ into at most $\sum_{i=0}^n \binom{k}{i} = O(n)k^n$ regions, and hence the same bound holds for C. □

Now some standard calculations imply Theorem 4.3.17, and hence Theorem 4.3.12.

Proof of Theorem 4.3.17. Lemma 4.3.19, together with the fact that there are at most $(k+1)^n$ cells, implies that the number of nonempty regions m is at most $O(n)k^n \cdot (k+1)^n \leq O(n)(k+1)^{2n}$. Since $m = 2^k$, this implies that $2^k \leq O(n)(k+1)^{2n}$. Taking logarithms of both sides yields that

$$k \leq \log(k+1) \cdot O(n), \tag{4.3}$$

and then taking another logarithm and rearranging yields that $\log n \geq \Omega(\log k - \log\log k) = \Omega(\log k)$ and hence $\log(k+1) \leq O(\log n)$. Plugging this into (4.3) implies Theorem 4.3.17. □

Bounding the Range

In this section we verify the condition for Theorem 4.3.14 that every function in $\mathcal{H}$ has its range in $[0, H]$ for $H = O(nC)$. This is actually not quite true as defined, but it is easy enough to ensure: we just consider a restricted class of functions $\mathcal{H}' = \{g_y \mid g_y \in \mathcal{H}, y \in [-C, C]^V\}$. Note that for any fixed set of costs c the class $\mathcal{H}'$ contains $y^*(c)$, so without loss of generality we can just use $\mathcal{H}'$ instead of $\mathcal{H}$. From the definition of pseudo-dimension and $\mathcal{H}' \subseteq \mathcal{H}$, it immediately follows that the pseudo-dimension of $\mathcal{H}'$ is at most that of $\mathcal{H}$. Thus, we just need to ensure that the range of the restricted functions are bounded.

Lemma 4.3.20. *Each function $g_y \in \mathcal{H}'$ has its range in $[0, H]$ for $H = O(nC)$.*

Proof. Let's bound the range by considering the maximum value g_y can take on a set of costs c. Recall that $g_y(c) = \|y - y^*(c)\|_1$. Each coordinate can contribute at most $O(C)$ to the sum since $y_i^*(c) \in [-C, C]$ and $y_i \in [-C, C]$. Summing over the n coordinates gives $H = O(nC)$. □

Minimizing the Empirical Loss

Now we give an algorithm to minimize the empirical loss on a collection of sample instances. Let $c_1, c_2, \ldots, c_s$ be a collection of samples from $\mathcal{D}$. Our goal is to find dual prices y minimizing $\frac{1}{s}\sum_{i=1}^{s} g_y(c_i) = \frac{1}{s}\sum_{i=1}^{s} \|y - y^*(c_i)\|_1$. Let $x^i = y^*(c_i)$. Then the problem amounts to minimizing $\frac{1}{s}\sum_{i=1}^{s} \|y - x^i\|_1$ over $y \in [-C, C]^V$. Then, for each coordinate j it suffices to find y_j minimizing $\sum_{i=1}^{s} \|y_j - x_j^i\|_1$, where y_j and x_j^i denote the j-th coordinate of y and x_i, respectively. Further, it is easy to see that $\sum_{i=1}^{s} \|y_j - x_j^i\|_1$ is a continuous piece-wise linear function in y_j where the slope can change only at $\{x_j^i\}_{i\in[s]}$. Recalling that we can assume wlog that $x^i = y^*(c_i)$ is an integer vector, we only need to consider setting y_j to each value in $\{x_j^i\}_{i\in[s]}$, which is a set of integers. This leads to the following result.

Theorem 4.3.21. *Given s samples $c_1, c_2, \ldots, c_s$, there exists a polynomial time algorithm which finds integer dual prices y minimizing $\frac{1}{s}\sum_{i=1}^{s} \|y - y^*(c_i)\|_1$.*

We remark that minimizing this emprical loss can be efficiently implemented by taking the coordinate-wise median of each optimal dual, i.e. taking $y_j = \text{median}(x_j^1, x_j^2, \ldots, x_j^s)$ for each $j \in V$.

4.4 Experiments

In this section we present experimental results on both synthetic and real data sets. Our goal is to validate the two main hypotheses in this work. First we show that warm-starting the Hungarian algorithm with learned duals provides an empirical speedup. Next, we show that the sample complexity of learning good duals is small, ensuring that our approach is viable in practice.

Experiment Setup: All of our experiments were run on Google Cloud Platform [73] `e2-standard-2` virtual machines with 2 virtual CPU's and 8 GB of memory.

Dataset	Blog Feedback[1]	Covertype	KDD	Skin[2]	Shuttle
# of Points (n)	52,397	581,012	98,942	100,000	43500
# of Features (d)	281	54	38	4	10

Table 4.1: Datasets used in experiments based on Euclidean data

We consider two different setups for learning dual variables and evaluating our algorithms.

- Batch: In this setup, we receive s samples $c_1, c_2, \ldots, c_s$ from the distribution of problem instances, learn the appropriate dual variables, and then test on new instances drawn from the distribution.

- Online: A natural use case for our approach is an *online* setting, where instance graphs $G_1, G_2, \ldots$ arrive one at a time. When deciding on the best warm start solution for G_t we can use all of the data from $G_1, \ldots, G_{t-1}$. This is a standard scenario in industrial applications like ad matching, where a new ad allocation plan may need to be computed daily or hourly.

Datasets: To study the effect of the different algorithm parameters, we first run a study on synthetic data. Let n be the number of nodes on one side of the bipartition and let ℓ, v be two parameters we set later. First, we divide the n nodes on each side of the graph into ℓ groups of equal size. The weight of all edges going from the i'th group on the left side and the j'th group on the right side is initialized to some value $W_{i,j}$ drawn from a geometric distribution with mean 250. Then to generate a particular graph instance, we perturb each edge weight with independent random noise according to a binomial distribution, shifted and scaled so that it has mean 0 and variance v. We refer to this as the *type* model (each type consists of a group of nodes). We use $n = 500$, $\ell \in \{50, 100\}$ and vary v from 0 to 2^{20}.

We use the following model of generating instances from real data. Let X be a set of n points in $\mathbb{R}^d$, and fix a parameter k. We first divide X randomly into two sets, X_L and X_R and compute a k-means clustering on each partition. To generate an instance $G = (L \cup R, E)$, we sample one point from each cluster on each side, generating $2k$ points in total. The points sampled from X_L (resp. X_R) form the vertices in L (resp. R). The weight of an (i, j) edge is the Euclidean distance between these two points. Changing k allows us to control the size of the instance.

We use several datasets from the UCI Machine Learning repository [106]. See Table 4.1 for a summary. For the KDD and Skin datasets we used a sub-sample of the original data (sizes given in Table 4.1).

Implemented Algorithms and Metrics: We implemented the Hungarian Algorithm (a particular instantiation of Algorithm 11, as discussed in Section 4.3.2) allowing for arbitrary seeding of a feasible integral dual. We experimented with having initial dual of 0 (giving the standard Hungarian Algorithm) as the baseline and having the initial duals come from our learning algorithm followed by Algorithm 10 to ensure feasibility (which we refer to as "Learned Duals"). We also added the following "tightening" heuristic, which is used in all standard implementations of the Hungarian algorithm: given any feasible dual solution y, set $y_i \leftarrow y_i + \min_{j \in N(i)}\{c_{ij} - y_i - y_j\}$ for all nodes i on one side of the bipartition. This can be quickly carried out in $O(n+m)$ time, and guarantees that each node on that side has at least one edge in E'. We compare the runtime and number of primal-dual iterations, reporting mean values and error bars denoting 95% confidence intervals. Running time results can also be found below.

To learn initial duals we use a small number of independent samples of each instance type. We compute an optimal dual solution for each instance in the sample. To combine these together into a single dual solution, we compute the median value for each node's set of dual values. This is an efficient implementation of the empirical risk minimization algorithm from Section 4.3.3.

Results: First, we examine the performance of Learned Duals in the batch setting described above. For these experiments, we used 20 training instances to learn the initial duals and then tested those on 10 new instances. For the type model, we used $\ell = 50$ and considered varying the variance parameter v. The left plot in Figure 4.1 shows the results as we increase v from 0 to 300. We see a moderate improvement in this case, even when the noise variance is larger than the mean value of an edge weight. Going further, in the middle plot of Figure 4.1 we consider increasing the noise variance in powers of two geometrically. Note that even when the noise significantly dominates the original signal from the mean weights (and hence the training instances should not help on the test instances), our method is comparable to the Hungarian method.

Continuing with the Batch setting, the right plot in Figure 4.1 summarizes our results for the clustering derived instances on all datasets with $k = 500$ (similar results hold for other values of k; see Figure 4.4). We see an improvement across all datasets, and a greater than 2x improvement on all but the Covertype dataset.

Figures 4.2 and 4.3 display our results in the online setting. We aim to show that not too many samples are needed to learn effective duals. From left to right, the plots in Figure 4.2 show the performance averaged over 20 repetitions of the experiment with 20 time points on the type model with $\ell = 100, v = 200$, and the clustering derived instances on the KDD and Covertype datasets with $k = 500$, respectively. We see that only a few iterations are needed to see a significant separation between the run time of our method with learned duals and the standard Hungarian method, with further steady improvement as we see more instances.

We see similar trends in both the real and synthetic data sets. We conclude the following.

- The theory is predictive of practice. Empirically, learning dual variables can lead to significant speed-up. This speed-up is achieved in both the batch and online settings.

- As the distribution is more concentrated, the learning algorithm performs better (as one would suspect).

- When the distribution is not concentrated and there is little to learn, then the algorithm has performance similar to the widely used Hungarian algorithm.

All together, these results demonstrate the strong potential for improvements in algorithm run time using machine-learned predictions for the weighted matching problem.

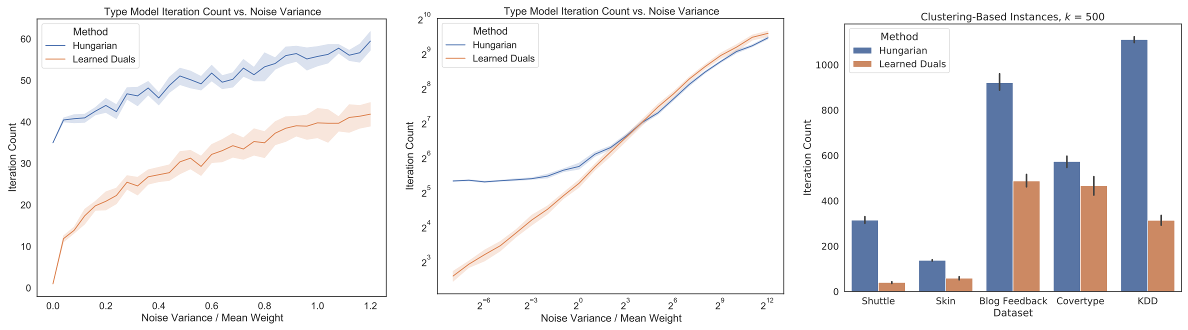

Figure 4.1: Iteration count results for the Batch setting. The left figure gives the iteration count for the type model (synthetic data) versus linearly increasing v, while the middle geometrically increases v. The right figure summarizes the results for clustering based instances (real data) in the batch setting.

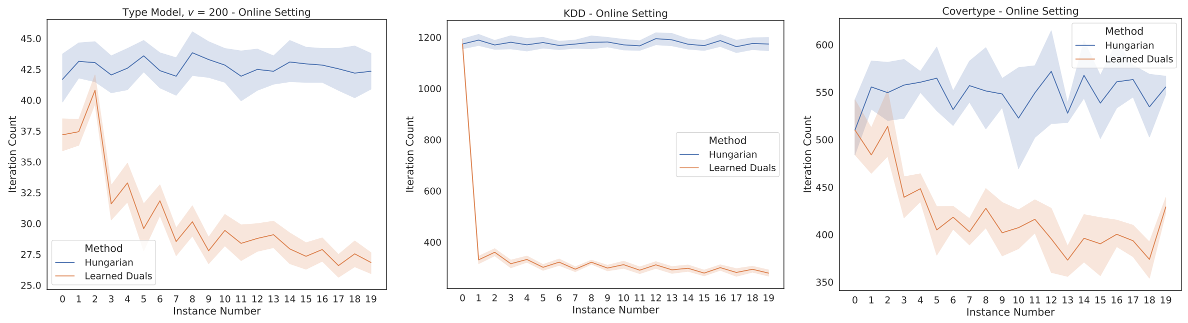

Figure 4.2: Iteration count results for the Online setting. The left figure is for the type model (synthetic data), while the middle and right are for the clustering based instances (real data) with $k = 500$ on KDD and Covertype, respectively.

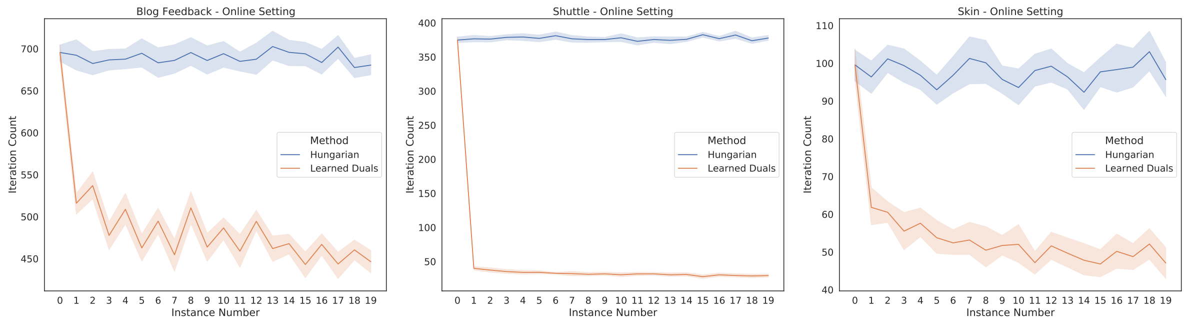

Figure 4.3: More iteration count results for the Online setting. From left to right, we have the results for the clustering based instances on Blog Feedback, Shuttle, and Skin, all with $k = 500$.

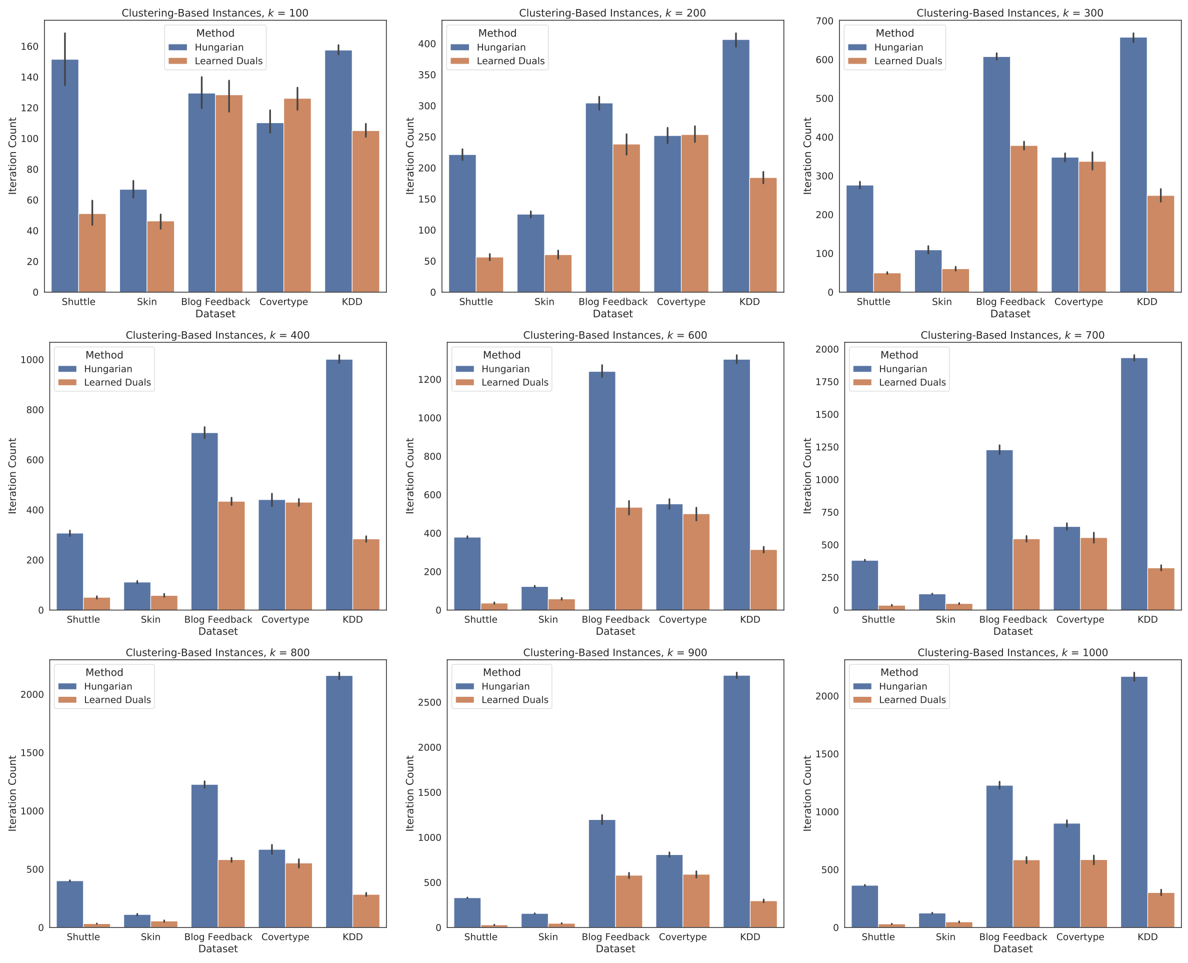

Figure 4.4: Iteration count results for clustering derived instances in the Batch setting on other values of k. Here we give the results for each k in $\{100 \cdot i \mid 1 \leq i \leq 10\} \setminus \{500\}$.

4.4.1 Running Time

Now we present our running time results (as opposed to number of primal dual iterations). We observe that there is a significant improvement for our method with respect to this metric over the standard initialization for the Hungarian algorithm.

Figure 4.5 gives running time results for the batch setting, while Figure 4.6 give the results for the online setting. Finally, Figure 4.7 looks at the clustering derived instances for other values of k. We see similar performance improvements for Learned Duals against the standard Hungarian algorithm, showing that the impact of running Algorithm 10 to make the predicted duals feasible is minimal.

4.5 Extending to b-Matching

We now extend the results from Section 4.3 to the minimum weight perfect b-matching problem on bipartite graphs. In the extension we are given a bipartite graph $G = (V, E)$,

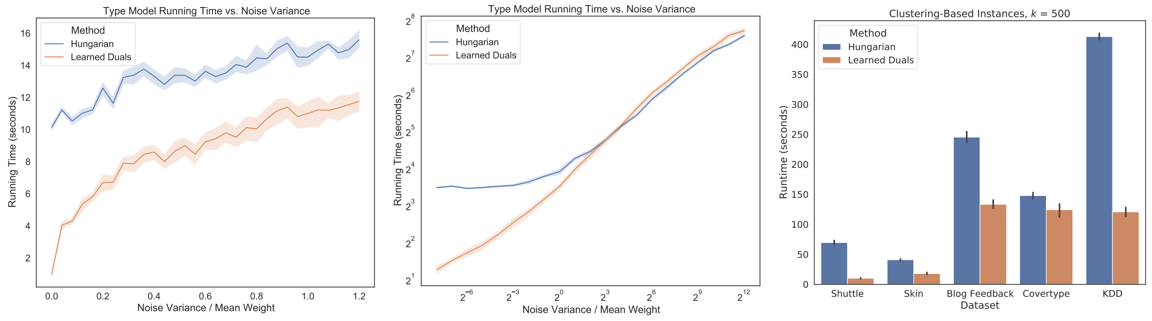

Figure 4.5: Running time results (in seconds) for the Batch setting. The left figure gives the iteration count for the type model (synthetic data) versus linearly increasing v, while the middle geometrically increases v. The right figure summarizes the results for clustering based instances (real data) in the batch setting.

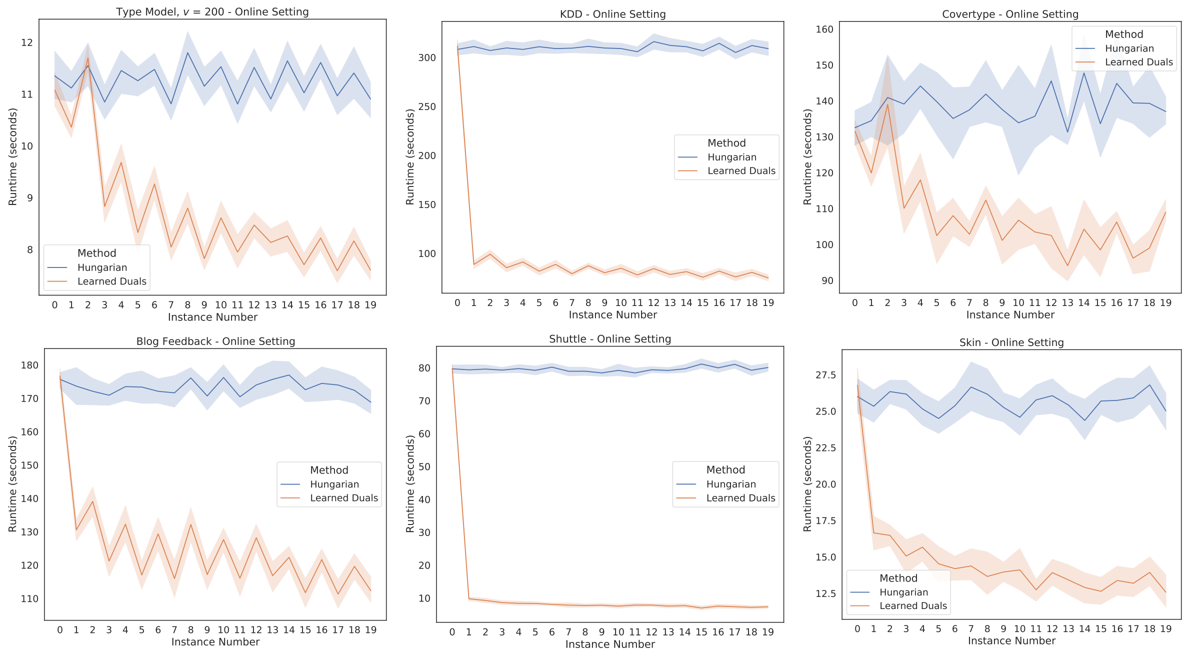

Figure 4.6: Running time results for the Online setting. The top left figure is for the type model (synthetic data). The rest, in order, are KDD and Covertype, Blog Feedback, Shuttle, and Skin. All use $k = 500$.

where $V = L \cup R$, a weight vector $c \in \mathbb{Z}_+^E$ and a demand vector $b \in \mathbb{Z}_+^V$. As before, we assume that the primal is feasible for the remainder of this section. Note that the feasibility of the primal can be checked with a single call to a maximum flow algorithm.

The problem is modeled by the following linear program and its dual linear program.

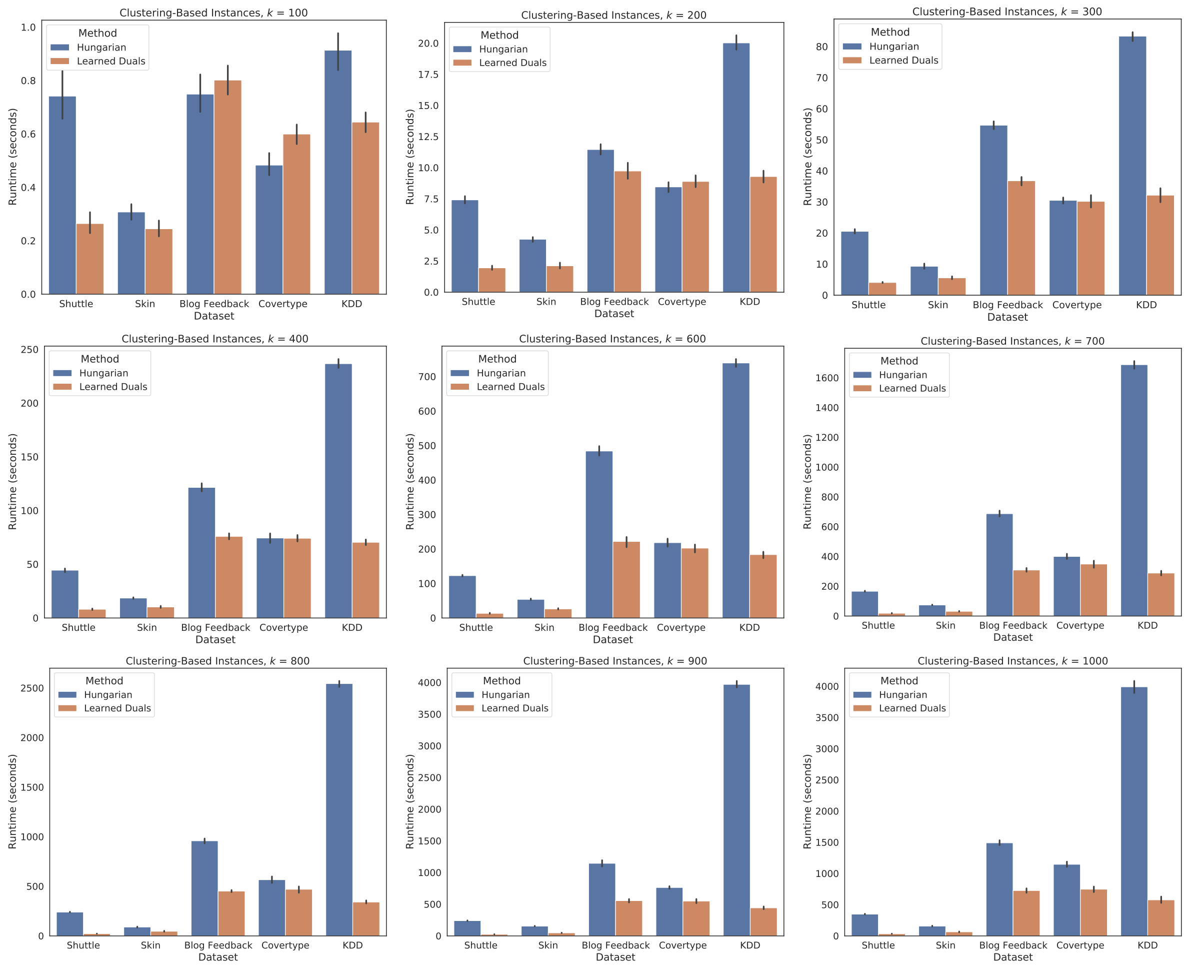

Figure 4.7: Running time results (in seconds) for clustering derived instances in the Batch setting on other values of k. Here we give the results for each k in $\{100 \cdot i \mid 1 \leq i \leq 10\} \setminus \{500\}$.

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ & \sum_{e \in \delta(i)} x_e = b_i \quad \forall i \in V \\ & x_e \geq 0 \qquad \forall e \in E \end{aligned} \tag{MWBM-P}$$

$$\begin{aligned} \max \quad & \sum_{i \in V} b_i y_i \\ & y_i + y_j \leq c_{ij} \quad \forall ij \in E \end{aligned} \tag{MWBM-D}$$

First we show how to project an infeasible dual onto the set of feasible solutions, then we give a simple primal dual scheme for moving to an optimal solution. The end goal of this section is proving the following theorem.

Theorem 4.5.1. *There exists an algorithm which takes as input a (not necessarily feasible) dual assignment y and finds a minimum weight perfect b-matching in $O(mn\|y^* - y\|_1)$ time, where y^* is an optimal dual solution and $\|y^* - y\|_{b,1} := \sum_i b_i |y_i^* - y_i|$.*

4.5.1 Recovering a Feasible Dual Solution for b-Matching

As in Section 4.3, our goal now is to find non-negative perturbations δ such that $\hat{y}' := \hat{y} - \delta$ is feasible for (MWPM-D). We would like these perturbations to preserve as much of the dual objective value as possible. Again we define $r_e := \hat{y}_i + \hat{y}_j - c_e$ for each edge $e = ij \in E$. Following the same steps as before, this leads to the following linear program and it's dual.

$$\begin{array}{lll} \min & \sum_{i \in V} b_i \delta_i & \\ & \delta_i + \delta_j \geq r_e & \forall e = ij \in E \\ & \delta_i \geq 0 & \forall i \in V \end{array} \tag{4.4}$$

$$\begin{array}{lll} \max & \sum_{e \in E} r_e \gamma_e & \\ & \sum_{e \in N(i)} \gamma_e \leq b_i & \forall i \in V \\ & \gamma_e \geq 0 & \forall e \in E \end{array} \tag{4.5}$$

Again we are interested in finding a fast approximate solution to this problem. We develop a new algorithm different than that used in the prior section and show it is a 2 approximation to (4.4). To do so, consider the dual LP above. This is an instance of the weighted b-matching problem where edges can be selected any number of times. We will first develop a 2 approximation to this LP in $O(m \log m + n)$ time. The analysis will be done via a dual fitting analysis. This analysis will give us the corresponding 2-approximate fractional primal solution that will be used to construct $\hat{y}'$.

Consider the following algorithm for the dual problem. Sort the edges e in decreasing order of r_e. When considering an edge $e' = i'j'$ in this order set $\gamma_{e'}$ as large as possible such that $\sum_{e \in N(i')} \gamma_e \leq b_{i'}$ and $\sum_{e \in N(j')} \gamma_e \leq b_{j'}$. Notice the running time of the algorithm is bounded by $O(m \log m + n)$.

When the algorithm terminates we construct a corresponding primal solutions. For each $i \in V$, set $\delta_i = \frac{\sum_{e \in N(i)} \gamma_e r_e}{b_i}$. That is, δ_i is the summation of the weights r of the adjacent edges divided by the b-matching constraint value b_i. We will show that δ is a feasible primal solution. Moreover that the primal and dual objectives are within a factor two of each other.

Lemma 4.5.2. *The solution δ is feasible for LP (4.4) and γ is feasible for the dual LP (4.5).*

Proof. The feasibility for the dual is by construction, so consider the primal. Consider any edge $e' = i'j'$. Our goal is to show that $\delta_{i'} + \delta_{j'} \geq r_{e'}$. Let $A_{e'}$ be the set of edges considered by the algorithm up to edge e' including the edge itself. These edges have weight at least as large e'. We claim that either $\sum_{e \in N(i') \cap A_{e'}} \gamma_e = b_{i'}$ or $\sum_{e \in N(j') \cap A_{e'}} \gamma_e = b_{j'}$. Indeed, otherwise we would increase $\gamma_{e'}$ until this is true. Without loss of generality say that $\sum_{e \in N(i') \cap A_{e'}} \gamma_e = b_{i'}$. We will argue that $\delta_{i'} \geq r_{e'}$. Knowing that $\delta_{j'}$ is non-negative, this will complete the proof.

Consider the value of $\delta_{i'}$. This is $\frac{\sum_{e \in N(i')} \gamma_e r_e}{b_{i'}} = \frac{\sum_{e \in N(i') \cap A_{e'}} \gamma_e r_e}{b_{i'}}$. We know from the above that $\sum_{e \in N(i') \cap A_{e'}} \gamma_e = b_{i'}$ and every edge in $A_{e'}$ has weight greater than e'. Thus, $\frac{\sum_{e \in N(i') \cap A_{e'}} \gamma_e r_e}{b_{i'}} \geq r_{e'} \frac{\sum_{e \in N(i') \cap A_{e'}} \gamma_e}{b_{i'}} = r_{e'}$ □

Next we bound the objective of the primal as a function of the dual.

Lemma 4.5.3. *The primal objective is exactly twice the dual objective.*

Proof. It suffices to show each edge $e = ij$ contributes twice as much to the primal objective as it does to the dual objective. First, e's contribution to the dual objective is clearly $r_e \gamma_e$. For the dual, edge e contributes to the summation for both end points. That is, e contributes to δ_i by $\gamma_e r_e / b_i$ and to δ_j by $\gamma_e r_e / b_j$. Thus, edge e's contribution to the primal objective is $b_i \frac{\gamma_e r_e}{b_i} + b_j \frac{\gamma_e r_e}{b_j} = 2\gamma_e r_e$, as desired. □

Thus, we have found a 2-approximate solution to the primal LP (4.4). However, the solution is not necessarily integral. Thus, to make it integral, we do the following simple rounding:

$$\delta_i \leftarrow \begin{cases} \lfloor 2\delta_i \rfloor & \text{if } \delta_i \geq 0.5 \\ 0 & \text{if } \delta_i \in [0, 0.5) \end{cases}$$

Clearly this update can double the cost in the worst case. Hence we only need to check that every constraint remains satisfied. To see this consider an edge $e = ij$ and let δ_i and δ_j be the dual values before the update. Note that r_e is an integer assuming that we are given integer dual values $\hat{y}$. Assume $r_e \geq 1$ since otherwise the constraint trivially holds true. It is an easy exercise to see that $\lfloor 2x \rfloor \geq x$ for all $x \geq 0.5$. Thus, if $\delta_i, \delta_j \geq 0.5$, then the update only increases the value of δ_i and δ_j, keeping the constraint satisfied. Further, as $r_e \geq 1$, it must be the case that $\delta_i \geq 0.5$ or $\delta_j \geq 0.5$. So, we only need to consider the case either $\delta_i \geq 0.5$ and $\delta_j < 0.5$; or $\delta_i < 0.5$ and $\delta_j \geq 0.5$. Assume wlog that the latter is the case. Since $\delta_i \leq \delta_j$, if $\delta_i + \delta_j \geq r_e$, we have $2\delta_j \geq r_e$. Then, we have $\lfloor 2\delta_j \rfloor \geq r_e$ as r_e is an integer. Again, the constraint is satisfied.

Thus, we obtain the following which is analogous to Theorem 4.3.5.

Theorem 4.5.4. *There is a $O(m \log m + n)$ time algorithm that takes an infeasible integer dual $\hat{y}$ and constructs a feasible integer dual $\hat{y}'$ such that $\|\hat{y} - \hat{y}'\|_{b,1} \leq 4\|y^* - \hat{y}\|_{b,1}$ where y^* is the optimal dual solution. Thus, we have $\|\hat{y}' - y^*\|_{b,1} \leq 5\|\hat{y} - y^*\|_{b,1}$.*

4.5.2 Converting a Feasible Dual Solution to an Optimal Primal Solution

Now we consider taking a feasible dual y and moving to an optimal solution for the b-matching problem. The algorithm we use is a simple primal-dual scheme that generalizes

Algorithm 11. See Algorithm 12 for details. Below we give a brief analysis of this algorithm. The objective is to establish a running time in terms of the following distance $\|y^* - y\|_{b,1} := \sum_i b_i|y_i^* - y_i|$. One can view this distance as the ℓ_1 norm distance where each coordinate axis is given a different level of importance by the b_i values.

Algorithm 12 Simple Primal-Dual Scheme for MWBM

```
 1: procedure MWBM-PrimalDual($G = (V, E), c, y$)
 2:     $E' \leftarrow \{ij \in E \mid y_i + y_j = c_{ij}\}$                  ▷ Set of tight edges in the dual
 3:     $G' \leftarrow (L \cup R \cup \{s, t\}, E' \cup \{si \mid i \in L\} \cup \{jt \mid j \in R\})$   ▷ Network of tight edges
 4:     $\forall e \in E(G')$ s.t. $e = si$ or $e = it$, $u_e \leftarrow b_i$
 5:     $u_e \leftarrow \infty$ for all other edges of $G'$
 6:     $f \leftarrow$ Maximum $s - t$ flow in $G'$ with capacities $u$
 7:     while Value of $f$ is $< \sum_{i \in L} b_i$ do
 8:         Find a set $S \subseteq L$ such that $\sum_{i \in S} b_i > \sum_{j \in \Gamma(S)} b_j$   ▷ Exists by Lemma 4.5.5
                                                                ▷ Can be found in $O(m + n)$ time
 9:         $\epsilon \leftarrow \min_{i \in S, j \in R \setminus \Gamma(S)}\{c_{ij} - y_i - y_j\}$
10:         $\forall i \in S$, $y_i \leftarrow y_i + \epsilon$
11:         $\forall j \in \Gamma(S)$, $y_j \leftarrow y_j - \epsilon$
12:         Update $E', G', u$
13:         $f \leftarrow$ Maximum $s - t$ flow in $G'$ with capacities $u$
14:     end while
15:     $x \leftarrow f$ restricted to edges of $G$
16:     Return $x$
17: end procedure
```

First we consider the correctness of the algorithm. As before, we need to show that the update rule is well defined. The following is a well known generalization of Hall's theorem, showing that line 8 is well defined. Further, the step can be implemented efficiently given f. The proof closely follows that of Proposition 4.3.6 – the only difference is factoring b in the matching size and vertex cover size.

Proposition 4.5.5. *Let G' be the flow network defined in Algorithm 12 with capacities ρ and let f be the maximum $s - t$ flow in G' if the value of f is less than $\sum_{i \in L} b_i$ then there exists $S \subseteq L$ such that $\sum_{i \in S} b_i > \sum_{j \in \Gamma(S)} b_j$. Further, such S can be found in $O(m + n)$ time.*

The following is analogous to Proposition 4.3.7 in Section 4.3.

Proposition 4.5.6. *Let y be dual feasible and suppose that $S \subseteq L$ with $\sum_{i \in S} b_i > \sum_{j \in \Gamma(S)} b_j$ in G'. Let $\epsilon = \min_{i \in S, j \in R \setminus \Gamma(S)}\{c_{ij} - y_i - y_j\}$. Then as long as c and y are integers we have $\epsilon \geq 1$.*

Additionally, we need to establish that y remains feasible throughout the execution of the algorithm. This is nearly identical to the corresponding lemma in Section 4.3 so we state it as the following lemma without proof.

Lemma 4.5.7. *If Algorithm 12 is given an initial dual feasible y, then y remains dual feasible throughout its execution.*

The above statements can be combined to give the following theorem.

Theorem 4.5.8. *There exists an algorithm for minimum weight perfect b-matching in bipartite graphs which runs in time $O(nm\|y^* - y\|_{b,1})$, where y^* is an optimal dual solution and y is the initial dual feasible solution passed to the algorithm.*

Proof. The correctness of the algorithm is implied by Lemma 4.5.7 and the fact that the flow network G' ensures that the resulting solution x that it finds satisfies complementary slackness with y. Thus we just need to establish the running time.

Note that it suffices to bound the number of iterations in terms of $O(\|y^* - y\|_{b,1})$ since the most costly step of each iteration is finding the maximum flow in the network G', which can be done in time $O(nm)$. The two propositions above state that the net increase in the dual objective is always at least 1, and so the number of iterations is at most $\sum_i b_i y_i^* - \sum_i b_i y_i \le \sum_i b_i \|y_i^* - y_i\| = \|y^* - y\|_{b,1}$. □

This theorem, combined with Theorem 4.5.4, gives Theorem 4.5.1, as desired.

4.5.3 Learning the Dual Prices

In this section we extend the results from Section 4.3.3 to the case of b-matching. As before, we consider a graph with fixed demands b and an unknown distribution $\mathcal{D}$ over the edge costs c. We are interested in learning a fixed set of prices y which is in some sense best for this distribution. Since the running time of the algorithms we consider depends on $\|y^* - y\|_{b,1}$ it is natural to choose this as our loss function with respect to the learning task. Thus we define $L_b(y,c) = \|y - y^*(c)\|_{b,1}$, where again $y^*(c)$ is a fixed optimal dual vector for costs c. Our goal is to perform well against the best choice for the distribution. Formally, let $y^* := \arg\min_y \mathbb{E}_{c\sim\mathcal{D}}[L_b(y,c)]$. Additionally, let C be a bound on the edge costs and $B = \max_{i\in V} b_i$ be a bound on the demands. We have the following result which is analogous to Theorem 4.3.12.

Theorem 4.5.9. *There is an algorithm that after $s = O\left(\left(\frac{nCB}{\epsilon}\right)^2 (n\log n + \log(1/\rho)\right)$ samples returns integer dual values $\hat{y}$ such that $\mathbb{E}_{c\sim\mathcal{D}}[L_b(\hat{y},c)] \le \mathbb{E}_{c\sim\mathcal{D}}[L(y^*,c] + \epsilon$ with probability at least $1-\rho$. The algorithm runs in time polynomial in n, m and s.*

At a high level, we can prove this theorem by again applying Theorem 4.3.14 and Corollary 4.3.15. To do this we define the following family of functions $\mathcal{H}_b = \{g_y \mid y \in \mathbb{R}^V\}$ where $g_y = \|y - y^*(c)\|_{b,1}$. We need to verify the following: (1) the range of these functions are bounded in $[0,H]$ for some $H = O(nCB)$, (2) minimizing the empirical loss can be done efficiently, and (3) the pseudo-dimension of $\mathcal{H}_b$ is bounded by $O(n\log n)$. Applying similar arguments as in Sections 4.3.3 and 4.3.3 give us the first two points. Here we focus on the last point, bounding the pseudo-dimension.

Note that for $b \in \mathbb{R}^n_+$, $\|\cdot\|_{b,1}$ is a norm. Intuitively, the geometry induced by $\|\cdot\|_{b,1}$ is the same as the geometry induced by $\|\cdot\|_1$ except some axes are stretched by an appropriate amount. This should imply that the functions in $\mathcal{H}_b$ should not be more complicated than the functions in $\mathcal{H}$. We make this intuition more formal by arguing that we can map from one setting to the other while preserving membership in the respective balls induced by these norms. The following key lemma will imply that the pseudo-dimension of $\mathcal{H}_b$ is no larger than the pseudo-dimension of $\mathcal{H}$.

Lemma 4.5.10. *Let $B_{b,1}(x, r) = \{y \mid \|x - y\|_{b,1} \leq r\}$ and $B_1(x, r) = \{y \mid \|x - y\|_1 \leq r\}$ be the balls of radius r under each norm, respectively. There is a mapping $\phi : \mathbb{R}^n \to \mathbb{R}^n$ such that $y \in B_{b,1}(x, r)$ if and only if $\phi(y) \in B_1(\phi(x), r)$.*

Proof. Define $\phi(y)_i = b_i y_i$ for $i = 1, 2, \ldots, n$. Now we have the following which implies the lemma.

$$\begin{aligned}\|x - y\|_{b,1} &= \sum_i b_i |x_i - y_i| = \sum_i |b_i x_i - b_i y_i| \\ &= \|\phi(x) - \phi(y)\|_1\end{aligned}$$

Thus one of these is at most r if and only if the other is. □

Now define the family of functions $\mathcal{H}_{b,n} = \{f_y : \mathbb{R}^n \to \mathbb{R} \mid y \in \mathbb{R}^n, f_y(x) = \|y - x\|_{b,1}\}$, we have the following which is analogous to Lemma 4.3.16.

Lemma 4.5.11. *The pseudo-dimension of $\mathcal{H}_b$ is at most the pseudo-dimension of $\mathcal{H}_{b,n}$*

Proof. Nearly identical to that of Lemma 4.3.16 but with $\|\cdot\|_1$ replaced with $\|\cdot\|_{b,1}$. □

We can now prove that the pseudo-dimension of $\mathcal{H}_b$ is bounded by $O(n \log n)$.

Lemma 4.5.12. *The pseudo-dimension of $\mathcal{H}_b$ is at most $O(n \log n)$.*

Proof. By Lemma 4.5.11 we have that the pseudo-dimension of $\mathcal{H}_b$ is at most $\mathcal{H}_{b,n}$. We now show that the pseudo-dimension of $\mathcal{H}_{b,n}$ is at most the pseudo-dimension of $\mathcal{H}_n$ using Lemma 4.5.10. Let $x^1, \ldots, x^k \in \mathbb{R}^n$ be given. Now consider $y^j = \phi(x^j)$ for $j = 1, \ldots, k$. By Lemma 4.5.10 we can see that $x^1, \ldots, x^k$ are shattered by $\mathcal{H}_{b,n}$ if and only if $y^1, \ldots, y^k$ are shattered by $\mathcal{H}_n$. Thus the pseudo-dimension of $\mathcal{H}_{b,n}$ is at most $\mathcal{H}_n$ and then the lemma follows by Theorem 4.3.17. □

4.6 Conclusion and Future Work

In this work we showed how to use learned predictions to warm-start primal-dual algorithms for weighted matching problems to improve their running times. We identified three key challenges of feasibility, learnability and optimization, for any such scheme, and showed that by working in the dual space we could give rigorous performance guarantees for each. Finally, we showed that our proposed methods are not only simpler, but also more efficient in practice.

An immediate avenue for future work is to extend these results to other combinatorial optimization problems. The key ingredient is identifying an appropriate intermediate representation: it must be simple enough to be learnable with small sample complexity, yet sophisticated enough to capture the underlying structure of the problem at hand.

Conclusion

This dissertation considered problems under the themes of scalability via *parallel and distributed algorithms* and *algorithms with predictions.*

In Chapters 1 and 2 we used approximation to give efficient parallel and distributed algorithms for weighted longest common subsequence and divisive hierarchical clustering, respectively. These are two widely used data analysis tasks for which there is a need to scale to larger inputs and standard methods are difficult to parallelize. At a high level, the use of approximations allowed us to overcome these difficulties.

In Chapters 3 and 4 we considered online makespan minimization with restricted assignment and minimum cost bipartite matchings in the setting of algorithms with predictions. In the former chapter, we define a prediction which consists of a positive weight for each machine which implicitly defines a fractional assignment which can be computed online. The fractional assignment gives a near optimal solution if the prediction is accurate, going beyond worst-case lower bounds for the problem

Potential Future Work

In terms of parallel and distributed algorithms, there is a potential to develop a better understanding of efficient distributed algorithms for hierarchical clustering. Much of the interest in hierarchical clustering in recent years has been due to the development of objective functions which try to capture properties of a "goood" hierarchical clustering [53, 146, 49, 147]. It is natural to aim for scalable algorithms which approximate these objectives. Indeed for some objectives such as those in [49] and [146], this can be done as it is known that the average-linkage algorithm gives an $O(1)$-approximation for these objectives and there are efficient, approximate implementations of average-linkage [3, 102].

Thus a relevant question is to develop scalable algorithms which approximate these other objectives. In particular, is there an $O(\text{poly}(\log n))$ round MPC algorithm which approximates the objective due to Dasgupta [53]? Standard sequential methods for this problem are based on reductions to sparsest cut or convex programming relaxations, which are difficult to adapt to the MPC setting.

In terms of algorithms with predictions, there are many possible directions to consider. One direction concerns the types of predictions that we learn. In some sense, this dissertation proposes learning a single "one size fits all" prediction from past problem instances which is then used to help solve new instances of the problem. A more practical, and potentially more effective approach might be to learn a mapping F, which takes in some instance specific features and outputs a prediction that is more instance-specific. This mapping would allow us to learn more complicated behaviour in the distribution over inputs. Indeed, this is what machine learning models for supervised learning problems attempt to do.

This direction may be challenging from both a theoretical and practical perspective, so a natural starting point is to restrict the types of mappings. One such restriction is to learn k different predictions from the past problem instances instead of a single prediction, then apply the best of these to the new problem instance. Of course, we may not immediately know which of the k predictions is the best for the new instance (in the online setting, for example), so there will be some trade-off between the quality of the k predictions and the performance of our algorithm as k increases. This question has been studied in the data-driven algorithm design literature [29], which has studied the sample complexity of learning k different predictions in several settings, but there are also interesting algorithmic questions for incorporating multiple predictions into the design of an algorithm and efficiently constructing the k different predictions from past problem instances. For example, given k different predicted weight vectors for online makespan minimization (as in Chapter 3), how do we aggregate the solutions given by these different predictions online in order to be competitive with the best one in hindsight?

Bibliography

[1] Anders Aamand, Justin Y. Chen, and Piotr Indyk. (optimal) online bipartite matching with predicted degrees, 2021.

[2] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Quadratic-time hardness of LCS and other sequence similarity measures. *CoRR*, abs/1501.07053, 2015.

[3] Amir Abboud, Vincent Cohen-Addad, and Hussein Houdrouge. Subquadratic high-dimensional hierarchical clustering. In *NeurIPS*, pages 11576–11586, 2019.

[4] A Aggarwal, MM Klawe, S Moran, P Shor, and R Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[5] Shipra Agrawal, Morteza Zadimoghaddam, and Vahab Mirrokni. Proportional allocation: Simple, distributed, and diverse matching with high entropy. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 99–108, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[6] Nir Ailon, Bernard Chazelle, Kenneth L. Clarkson, Ding Liu, Wolfgang Mulzer, and C. Seshadhri. Self-improving algorithms. *SIAM J. Comput.*, 40(2):350–375, 2011.

[7] CER Alves, EN Cáceres, and SW Song. A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica*, 45:301–335, 2006.

[8] CER Alves, EN Cáceres, and SW Song. An all-substrings common subsequence algorithm. *Discrete Applied Mathematics*, 156(7):1025 – 1035, 2008.

[9] Amazon Web Services, 2022.

[10] Keerti Anand, Rong Ge, Amit Kumar, and Debmalya Panigrahi. A regression approach to learning-augmented online algorithms. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[11] Keerti Anand, Rong Ge, and Debmalya Panigrahi. Customizing ML predictions for online algorithms. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 303–313. PMLR, 2020.

[12] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev. Parallel algorithms for geometric graph problems. In *STOC*, 2014.

[13] Martin Anthony and Peter L Bartlett. *Neural network learning: Theoretical foundations.* cambridge university press, 2009.

[14] Antonios Antoniadis, Christian Coester, Marek Eliás, Adam Polak, and Bertrand Simon. Online metric algorithms with untrusted predictions. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 345–355. PMLR, 2020.

[15] Antonios Antoniadis, Themis Gouleakis, Pieter Kleer, and Pavel Kolev. Secretary and online matching problems with machine learned advice. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[16] Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990.

[17] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[18] James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *J. ACM*, 44(3):486–504, May 1997.

[19] Yossi Azar, Andrei Z. Broder, and Mark S. Manasse. On-line choice of on-line algorithms. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 432–440. ACM/SIAM, 1993.

[20] Yossi Azar, Stefano Leonardi, and Noam Touitou. Flow time scheduling with uncertain processing time. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1070–1080. ACM, 2021.

[21] Yossi Azar, Joseph Seffi Naor, and Raphael Rom. The competitiveness of on-line assignments. *J. Algorithms*, 18(2):221–237, March 1995.

[22] Yossi Azar, Debmalya Panigrahi, and Noam Touitou. Online graph algorithms with predictions. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 35–66, 2022.

[23] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.*, 66(11):1366–1378, 2006.

[24] Maria-Florina Balcan. Data-driven algorithm design, 2020.

[25] Maria-Florina Balcan, Dan F. DeBlasio, Travis Dick, Carl Kingsford, Tuomas Sandholm, and Ellen Vitercik. How much data is sufficient to learn high-performing algorithms? *CoRR*, abs/1908.02894, 2019.

[26] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 353–362. PMLR, 2018.

[27] Maria-Florina Balcan, Travis Dick, and Ellen Vitercik. Dispersion for data-driven algorithm design, online learning, and private optimization. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 603–614. IEEE Computer Society, 2018.

[28] Maria-Florina Balcan, Travis Dick, and Colin White. Data-driven clustering via parameterized lloyd's families. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 10664–10674, 2018.

[29] Maria-Florina Balcan, Tuomas Sandholm, and Ellen Vitercik. Generalization in portfolio-based algorithm selection. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 12225–12232. AAAI Press, 2021.

[30] Eric Balkanski, Aviad Rubinstein, and Yaron Singer. The power of optimization from samples. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4017–4025, 2016.

[31] Étienne Bamas, Andreas Maggiori, and Ola Svensson. The primal-dual method for learning augmented algorithms. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[32] Nikhil Bansal, Christian Coester, Ravi Kumar, Manish Purohit, and Erik Vee. Learning-augmented weighted paging. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 67–89, 2022.

[33] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, 2016.

[34] Mohammadhossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. Affinity clustering: Hierarchical clustering at scale. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[35] MohammadHossein Bateni, Aditya Bhaskara, Silvio Lattanzi, and Vahab S. Mirrokni. Distributed balanced clustering via mapping coresets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2591–2599, 2014.

[36] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284. ACM, 2013.

[37] Rajen Bhatt and Abhinav Dhall. Skin segmentation dataset, uci machine learning repository, 2012.

[38] Joan Boyar, Lene M. Favrholdt, Christian Kudahl, Kim S. Larsen, and Jesper W. Mikkelsen. Online algorithms with advice: A survey. *SIGACT News*, 47(3):93–129, August 2016.

[39] Karl Bringmann and Bhaskar Ray Chaudhury. Sketching, streaming, and fine-grained complexity of (weighted) LCS. In *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India*, pages 40:1–40:16, 2018.

[40] Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1216–1235, 2018.

[41] Sébastien Bubeck and Aleksandrs Slivkins. The best of both worlds: Stochastic and adversarial bandits. In *COLT 2012 - The 25th Annual Conference on Learning Theory, June 25-27, 2012, Edinburgh, Scotland*, pages 42.1–42.23, 2012.

[42] Jeremy Buhler, Thomas Lavastida, Kefu Lu, and Benjamin Moseley. A scalable approximation algorithm for weighted longest common subsequence. In Leonel Sousa, Nuno Roma, and Pedro Tomás, editors, *Euro-Par 2021: Parallel Processing - 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1-3, 2021, Proceedings*, volume 12820 of *Lecture Notes in Computer Science*, pages 368–384. Springer, 2021.

[43] Krisztian Buza. Feedback prediction for blogs. In Myra Spiliopoulou, Lars Schmidt-Thieme, and Ruth Janning, editors, *Data Analysis, Machine Learning and Knowledge Discovery*, pages 145–152, Cham, 2014. Springer International Publishing.

[44] Deeparnab Chakrabarty, Sanjeev Khanna, and Shi Li. On (1,ϵ))-restricted assignment makespan minimization. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1087–1101, 2015.

[45] Moses Charikar and Vaggos Chatziafratis. Approximate hierarchical clustering via sparsest cut and spreading metrics. In *SODA*, 2017.

[46] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. *SICOMP*, 33(6):1417–1440, 2004.

[47] Moses Charikar, Sudipto Guha, Éva Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k-median problem. *J. Comput. Syst. Sci.*, 65(1):129–149, 2002.

[48] Antonia Chmiela, Elias Boutros Khalil, Ambros Gleixner, Andrea Lodi, and Sebastian Pokutta. Learning to schedule heuristics in branch and bound. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[49] Vincent Cohen-addad, Varun Kanade, Frederik Mallmann-trenn, and Claire Mathieu. Hierarchical clustering: Objective functions and algorithms. *J. ACM*, 66(4), 2019.

[50] Richard Cole and Tim Roughgarden. The sample complexity of revenue maximization. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 243–252, New York, NY, USA, 2014. ACM.

[51] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32:1654–1673, 09 2003.

[52] Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 451–460. ACM, 2008.

[53] Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In *STOC*, 2016.

[54] Nikhil R. Devanur and Thomas P. Hayes. The adwords problem: online keyword matching with budgeted bidders under random permutations. In *Proceedings 10th ACM Conference on Electronic Commerce (EC-2009), Stanford, California, USA, July 6–10, 2009*, pages 71–78, 2009.

[55] Ilias Diakonikolas, Vasilis Kontonis, Christos Tzamos, Ali Vakilian, and Nikos Zarifis. Learning online algorithms with distributional advice. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 2687–2696. PMLR, 2021.

[56] Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[57] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[58] Doratha E. Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, 2003.

[59] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1: 1–1: 23, 2014.

[60] Ran Duan and Hsin - Hao Su. A scaling algorithm for maximum weight matching in bipartite graphs. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1413–1424. SIAM, 2012.

[61] Paul Duetting, Zhe Feng, Harikrishna Narasimhan, David C. Parkes, and Sai Srivatsa Ravindranath. Optimal auctions through deep learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 1706–1715. PMLR, 2019.

[62] Paul Dütting, Silvio Lattanzi, Renato Paes Leme, and Sergei Vassilvitskii. Secretaries with advice. In Péter Biró, Shuchi Chawla, and Federico Echenique, editors, *EC '21:*

The 22nd ACM Conference on Economics and Computation, Budapest, Hungary, July 18-23, 2021, pages 409–429. ACM, 2021.

[63] Michael B. Eisen, Paul T. Spellman, Patrick O. Brown, and David Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 95(25):14863–14868, 1998.

[64] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using MapReduce. In *KDD*, pages 681–689, 2011.

[65] Steven Fortune and James Wyllie. Parallelism in random access machines. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pages 114–118. ACM, 1978.

[66] Harold N. Gabow. A scaling algorithm for weighted matching on general graphs. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 90–100. IEEE Computer Society, 1985.

[67] Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 270–277, 2016.

[68] Andrew V. Goldberg and Robert Kennedy. Global price updates help. *SIAM J. Discret. Math.*, 10(4):551–572, 1997.

[69] Sreenivas Gollapudi and Debmalya Panigrahi. Online algorithms for rent-or-buy with expert advice. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2319–2327. PMLR, 2019.

[70] Jacek Gondzio. Warm start of the primal-dual method applied in the cutting-plane scheme. *Math. Program.*, 83:125–143, 1998.

[71] Jacek Gondzio and Pablo González-Brevis. A new warmstarting strategy for the primal-dual column generation method. *Math. Program.*, 152(1-2):113–146, 2015.

[72] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2011.

[73] Google Cloud Platform. `https://cloud.google.com/`, 2022.

[74] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 18(1):54–64, 1969.

[75] Rishi Gupta and Tim Roughgarden. A PAC approach to application-specific algorithm selection. *SIAM J. Comput.*, 46(3):992–1017, 2017.

[76] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *Unsupervised Learning*, pages 485–585. Springer New York, New York, NY, 2009.

[77] Katherine A. Heller and Zoubin Ghahramani. Bayesian hierarchical clustering. In *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, pages 297–304, 2005.

[78] Dorit S. Hochbaum and David B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *J. ACM*, 33(3):533–550, 1986.

[79] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$-algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[80] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. In *7th International Conference on Learning Representations*, 2019.

[81] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 798–811, 2017.

[82] Piotr Indyk, Frederik Mallmann-Trenn, Slobodan Mitrovic, and Ronitt Rubinfeld. Online page migration with ML advice. *CoRR*, abs/2006.05028, 2020.

[83] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010.

[84] Klaus Jansen and Lars Rohwedder. On the configuration-lp of the restricted assignment problem. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2670–2678, 2017.

[85] Zhihao Jiang, Debmalya Panigrahi, and Kevin Sun. Online algorithms for weighted paging with predictions. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 69:1–69:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[86] Chen Jin, Zhengzhang Chen, William Hendrix, Ankit Agrawal, and Alok N. Choudhary. Incremental, distributed single-linkage hierarchical clustering algorithm using mapreduce. In *HPC*, 2015.

[87] Chen Jin, Ruoqian Liu, Zhengzhang Chen, William Hendrix, Ankit Agrawal, and Alok N. Choudhary. A scalable hierarchical clustering algorithm using spark. In *Big Data Computing Service and Applications*, 2015.

[88] Neil Jones. *An introduction to bioinformatics algorithms.* MIT Press, Cambridge, MA, 2004.

[89] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. *Comput. Geom.*, 28(2-3):89–112, 2004.

[90] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948. SIAM, 2010.

[91] Alexander V Karzanov. On finding maximum flows in networks with special structure and some applications. *Matematicheskie Voprosy Upravleniya Proizvodstvom*, 5:81–94, 1973.

[92] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447 – 474, 1994.

[93] Weiwei Kong, Christopher Liaw, Aranyak Mehta, and D. Sivakumar. A new dog learns old tricks: RL finds classic optimization algorithms. In *International Conference on Learning Representations*, 2019.

[94] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[95] Akshay Krishnamurthy, Sivaraman Balakrishnan, Min Xu, and Aarti Singh. Efficient active algorithms for hierarchical clustering. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*, 2012.

[96] Peter Krusche and Alexandre Tiskin. Efficient longest common subsequence computation using bulk-synchronous parallelism. In *Computational Science and Its Applications - ICCSA 2006, International Conference, Glasgow, UK, May 8-11, 2006, Proceedings, Part V*, pages 165–174, 2006.

[97] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[98] Ravi Kumar, Manish Purohit, Aaron Schild, Zoya Svitkina, and Erik Vee. 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, 2019.

[99] Silvio Lattanzi, Thomas Lavastida, Kefu Lu, and Benjamin Moseley. A framework for parallelizing hierarchical clustering methods. In Ulf Brefeld, Élisa Fromont, Andreas Hotho, Arno J. Knobbe, Marloes H. Maathuis, and Céline Robardet, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, volume 11906 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2019.

[100] Silvio Lattanzi, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Online scheduling via learned weights. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1859–1877. SIAM, 2020.

[101] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 85–94. ACM, 2011.

[102] Thomas Lavastida, Kefu Lu, Benjamin Moseley, and Yuyan Wang. Scaling average-linkage via sparse cluster embeddings. In Vineeth N. Balasubramanian and Ivor W. Tsang, editors, *Asian Conference on Machine Learning, ACML 2021, 17-19 November 2021, Virtual Event*, volume 157 of *Proceedings of Machine Learning Research*, pages 1429–1444. PMLR, 2021.

[103] Thomas Lavastida, Benjamin Moseley, R. Ravi, and Chenyang Xu. Learnable and instance-robust predictions for online matching, flows and load balancing. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 59:1–59:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[104] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in o (vrank) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433. IEEE Computer Society, 2014.

[105] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1):259–271, Jan 1990.

[106] M. Lichman. UCI ml repository, 2013.

[107] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 3302–3311, 2018.

[108] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *J. ACM*, 68(4):24:1–24:25, 2021.

[109] Mohammad Mahdian, Hamid Nazerzadeh, and Amin Saberi. Online optimization with uncertain information. *ACM Trans. Algorithms*, 8(1):2:1–2:29, 2012.

[110] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[111] Aranyak Mehta, Amin Saberi, Umesh V. Vazirani, and Vijay V. Vazirani. Adwords and generalized online matching. *J. ACM*, 54(5):22, 2007.

[112] Microsoft Azure. `https://azure.microsoft.com/en-us/`, 2022.

[113] Vahab S. Mirrokni, Shayan Oveis Gharan, and Morteza Zadimoghaddam. Simultaneous approximations for adversarial and stochastic online budgeted allocation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1690–1701, 2012.

[114] Michael Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems*, pages 464–473, 2018.

[115] Michael Mitzenmacher. Scheduling with predictions and the price of misprediction. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[116] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*, pages 646–662. Cambridge University Press, 2020.

[117] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[118] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[119] Vinod Nair, Dj Dvijotham, Iain Dunning, and Oriol Vinyals. Learning fast optimizers for contextual stochastic integer programs. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial*

Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018, pages 591–600. AUAI Press, 2018.

[120] SB Needleman and CD Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Molecular Biology*, 48:443–453, 1970.

[121] James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *opera. Res.*, 41(2):338–350, 1993.

[122] James B. Orlin. Max flows in o(nm) time, or better. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 765–774, New York, NY, USA, 2013. Association for Computing Machinery.

[123] James B. Orlin and Ravindra K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *math. Program.*, 54:41–56, 1992.

[124] David Pollard. *Convergence of stochastic processes.* Springer Science & Business Media, 2012.

[125] NHGRI Genome Sequencing Program. DNA sequencing costs: Data, 2017. `https://www.genome.gov/sequencingcostsdata/`.

[126] Manish Purohit, Zoya Svitkina, and Ravi Kumar. Improving online algorithms via ML predictions. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 9684–9693, 2018.

[127] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, Dec 2004.

[128] Dhruv Rohatgi. Near-optimal bounds for online caching with machine learned advice. In *Symposium on Discrete Algorithms (SODA)*, 2020.

[129] Tim Roughgarden. *Beyond the Worst-Case Analysis of Algorithms.* Cambridge University Press, 2020.

[130] Aurko Roy and Sebastian Pokutta. Hierarchical clustering via spreading metrics. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2316–2324, 2016.

[131] LMS Russo. Monge properties of sequence alignment. *Theor. Comput. Sci.*, 423:30–49, 2012.

[132] `http://jaligner.sourceforge.net/`.

[133] `https://github.com/mengyao/Complete-Striped-Smith-Waterman-Library`.

[134] `https://github.com/Martinsos/opal`.

[135] J Schmidt. All highest scoring paths in weighted graphs and their applications to finding all approximate repeats in strings. *SIAM J. Comput.*, 27:972–992, 1998.

[136] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62:461–474, 1993.

[137] TF Smith and MS Waterman. Identification of common molecular subsequences. *J. Molecular Biology*, 147(1):195–197, 1981.

[138] DJ States, W Gish, and SF Altschul. Improved sensitivity of nucleic acid database searches using application-specific scoring matrices. *METHODS: a companion to Methods in Enzymology*, 3:66–70, 1991.

[139] Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In *In KDD Workshop on Text Mining*, 2000.

[140] Ola Svensson. Santa claus schedules jobs on unrelated machines. *SIAM J. Comput.*, 41(5):1318–1341, 2012.

[141] A Tiskin. Semi-local string comparison: algorithmic techniques and applications, ch 6.1. https://arxiv.org/abs/0707.3619, 2013. v21.

[142] A Tiskin. Fast distance multiplication of unit-monge matrices. *Algorithmica*, 71(4):859–888, Apr 2015.

[143] Jan van den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and ℓ_1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021.

[144] Jan van den Brand, Yin-Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 919–930. IEEE, 2020.

[145] Erik Vee, Sergei Vassilvitskii, and Jayavel Shanmugasundaram. Optimal online assignment with forecasts. In David C. Parkes, Chrysanthos Dellarocas, and Moshe Tennenholtz, editors, *Proceedings 11th ACM Conference on Electronic Commerce (EC-2010), Cambridge, Massachusetts, USA, June 7-11, 2010*, pages 109–118. ACM, 2010.

[146] Joshua Wang and Benjamin Moseley. Approximation bounds for hierarchical clustering: Average-linkage, bisecting k-means, and local search. In *NIPS*, 2017.

[147] Yuyan Wang and Benjamin Moseley. An objective for hierarchical clustering in euclidean space and its connection to bisecting k-means. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 6307–6314. AAAI Press, 2020.

[148] Alexander Wei. Better and simpler learning-augmented online caching. In Jaroslaw Byrka and Raghu Meka, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2020, August 17-19, 2020, Virtual Conference*, volume 176 of *LIPIcs*, pages 60:1–60:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[149] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, 2011.

[150] Chenyang Xu and Benjamin Moseley. Learning-augmented algorithms for online steiner tree. *CoRR*, abs/2112.05353, 2021.

[151] Hiroshi Yamashita and Takahito Tanabe. A primal-dual exterior point method for nonlinear optimization. *SIAM Journal on Optimization*, 20(6):3335–3363, 2010.

[152] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 222–227. IEEE Computer Society, 1977.

[153] Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under lp distances. In *ICML*, 2018.