TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

The Raymond and Beverly Sackler Faculty of Exact Sciences

The Blavatnik School of Computer Science

Designing Local Computation Algorithms and Mechanisms

Thesis submitted for the degree of Doctor of Philosophy

by

Shai Vardi

This work was carried out under the supervision of

Professor Yishay Mansour

Submitted to the Senate of Tel Aviv University

September 2015

This work is dedicated to June,

born just in time to receive this dedication instead of her mom.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Yishay Mansour for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to give a special thank you to Ronitt and Noga for being amazing people and for having taught me so much. But mostly for being amazing people. I am blessed to know you.

I want to thank my co-authors - Alex, Aviad, Avinatan, Boaz, Eric, Ning, Noga, Omer, Ronitt and Yishay - I have learned a lot from each of you. I am also happy to call each one of you my friend.

Unfortunately I didn't get to collaborate with everyone I would have liked to. In particular: Amos, Michal, Wojciech and Yossi; Alan, Alon, Dan, Iddan, Ilan, Ophir and Yaniv. I hope we'll get to work together sometime! I would also like to thank those who made my life at the university and at conferences so much fun (most of you mentioned above, some aren't; you know who you are).

Thanks Pnina, Diana and Larisa for making my student life easy. Thanks Tami for listening. Thanks Hagai, Or and Yehuda for being such good friends. Thank you Google for the generous fellowship. If you feel I have forgotten you, you are probably right, and I'm sorry.

Thank you to all my reviewers, especially to reviewers 1 and 2.

Thank you for reading this entire acknowledgments section. You really didn't have to, and to be honest, I'm kind of surprised you reached this far. But this means that I am important to you in some way, and as a token of my gratitude, please accept this coupon for one cup of coffee. To redeem it, send me an email with the code word: Alaska.

I want to thank my family: my mother, father and brother for their support throughout.

Most importantly, I want to thank Shiri, who has always been there for me, always believed in me, and is pretty much the reason I am the person I am today. You are amazing!

As for the ants, you were disappointing. Bad ants. Bants.

Abstract

Local computation algorithms (LCAs) produce small parts of a single solution to a given search problem using time and space polylogarithmic in the size of the input. Consider, for example, a massive graph on which we would like to compute a maximal independent set (MIS). The graph is so big that computing the entire solution would simply take too long, and we do not have enough space to store the solution. Assume, though, that we never need the entire solution at any given time. Instead, we are occasionally queried about whether certain vertices are in the MIS. An LCA would allow us to reply to these queries consistently (i.e., if the LCA is queried on all of the vertices the replies would conform to a coherent solution), using polylogarithmic time and space. In this work we study several techniques for designing LCAs, and provide some useful tools for future research.

We introduce a new graph family, d-light graphs, that includes graphs of constant bounded degree and several interesting randomly generated graphs, such as Erdős-Rényi graphs, and bipartite graphs where vertices on one side are linked to d vertices on the other, chosen uniformly at random. We show how to reduce LCAs for certain problems on this graph family to distributed and online algorithms.

The reduction to distributed algorithms is due to Parnas and Ron [86]; its analysis on d-light graphs is new. We derive a reduction to online algorithms that is based on an idea by Onak and Nguyen [79]: generate a random order on the vertices and simulate the online algorithm on this order. We show a small seed of length $O(\log n)$ is sufficient to generate the randomness we need, and that with high probability, we do not require more than $O(\log^2 n)$ time or space to reply to each query. These reductions allow us to obtain LCAs for problems such as MIS, maximal matching, load balancing and vertex coloring on d-light graphs. We also extend these techniques to obtain a $(1-\epsilon)$-approximation LCA to maximum matching on graphs of constant bounded degree.

We further show that in some cases, we can design LCAs whose running time (and

space) is independent of the size of the graph, and depends only on the maximal degree. Specifically, we give a $(1-\epsilon)$-approximation LCA to the maximal weighted base of a graphic matroid (i.e., maximal acyclic edge set); LCAs for approximating multicut and integer multicommodity flow on trees; and a local reduction of weighted matching to any unweighted matching LCA, such that the running time of the weighted matching LCA is independent of the edge weight function.

The field of mechanism design involves designing algorithms for strategic environments - ones in which input to the algorithm consists of (or includes) private information divulged by agents who have a stake in the outcome. We introduce *local computation mechanism design* - designing mechanisms that are queried on a single agent, and are required to return the part of the solution that is relevant to her (the agent), while preserving the required global game theoretic properties. As an example, consider an auction of millions of items to millions of agents. When queried on an agent, we would like our mechanism to tell us which items she gets and how much she has to pay, and at the same time incentivize all agents to be truthful about their valuations for the items. Similarly to LCAs, local computation mechanisms reply to each query in polylogarithmic time and space, and the replies to different queries are consistent with the same global feasible solution. When the mechanism employs payments, the computation of the payments is also done in polylogarithmic time and space.

We present local computation mechanisms for a variety of classical game-theoretical problems: (1) stable matching, (2) job scheduling, (3) combinatorial auctions for unit-demand and k-minded bidders, and (4) the housing allocation problem.

For stable matching, some of our techniques have implications to the global (non-LCA) setting. Specifically, we show that when the men's preference lists are bounded, we can achieve an arbitrarily good approximation to the stable matching within a fixed number of iterations of the Gale-Shapley algorithm.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Solving combinatorial problems on graphs, such as maximal matching and vertex coloring, has been at the heart of computer science research since the first half of the previous century (e.g., [16, 40]), well before computers became household objects. Although we would generally like to solve any problem as fast as possible, we would also like a notion of what it means for a problem to be solvable "quickly". The most widely accepted definition of "quickly" in this context is *polynomial time* (see, e.g., [22, 50]). In the 1970s, it was shown that many combinatorial problems cannot be solved in polynomial time, unless $P = NP$ [50]. This inspired research devoted to what *can* be done in polynomial time, which includes finding polynomial-time approximation algorithms to some of these NP-hard problems (for a survey of approximation algorithms, see [107]). For many years, research in algorithms focused on what can and cannot be done in polynomial time, but in the past two decades, many computer systems have become so big that polynomial-time tractability is not enough. An algorithm that runs in $O(n^3)$ will simply not be practical on a graph with 1 billion nodes. A few decades ago, such graphs were rare, but today, they are ubiquitous: the Internet has over 4 billion indexed pages, and an estimated 45+ billion pages overall [1]; social networks have become huge: Facebook has over 1 billion users [111]; astronomical, biological, environmental, finance and sensor network data is often measured in petabytes (10^{15} bytes) [43].

The rapid growth in the size of networks and databases over the past decade or so has inspired a large volume of work on handling the challenges posed by these massive systems - systems so large, in fact, that algorithms cannot even be required to run in linear time. Much of the research on these problems therefore focuses on what can

be done in sublinear time. Despite this common denominator, the approaches have been diverse. The field of property testing [37, 91] asks whether some mathematical object, such as a graph, has a certain property, or is "far" from having that property; streaming algorithms [73, 113] are required to use limited memory and quickly process data streams that are presented as a sequence of items; sublinear approximation algorithms (e.g., [79, 86, 97]) give approximate solutions to optimization problems. All of the above examples compute some approximation to a solution: property testing algorithms usually reply "yes/no"; streaming algorithms usually compute some statistic on the input; sublinear approximation algorithms usually return an integer that is an estimate of the size of a solution. *Local computation algorithms* (LCAs) offer a different approach: instead of computing a value (or values) that give information about a feasible solution, we would like to actually compute some small part of the solution.

The need for LCAs arises in situations when we require fast and space-efficient access to part of a solution of a computational problem, but we never need the entire solution at once. Consider, for instance, a wireless network with millions of nodes. If two neighbors broadcast at the same time, this will cause interference; therefore we would like to schedule broadcasts such that neighboring nodes do not broadcast at the same time. A customary solution to this problem is to compute a coloring of the graph - each node is allocated a color, so that no two adjacent nodes have the same color, and all nodes of the same color broadcast at the same time. Assume now that each node may query when to broadcast (i.e., what its color is), but we are never required to compute the entire solution at once; in fact, as accessing shared memory is expensive, we do not even want to store any part of the solution that we may have already computed. Instead, we are allocated space polylogarithmic in the number of nodes, and are required to reply to each query in polylogarithmic time. Although at any single point in time, we can be queried the color of a single node, over a longer time period, we may be queried about all of the nodes. We therefore want all of our replies to be consistent with a feasible coloring.

LCAs are also useful in other scenarios: if uncoordinated processors are required to compute consistent results on a very large data set, LCAs immediately offer a solution in which the processors need access to a very small amount of shared memory (or in some cases, none at all), and they can reply to queries quickly and consistently. Another interesting non-trivial application of LCAs is the following. Assume that we have a

linear program with an exponential number of variables and constraints. However, we are only interested in the value of a polynomial number of variables in some "good"[1] solution. LCAs can allow us to obtain our required solution in polynomial time.

1.1 Local Computation Algorithms

Local computation algorithms were formally introduced by Rubinfeld et al. [98], although by then quite a large volume of work had been devoted to LCAs and very closely related topics. A well-known example is the computation of the relative importance of web pages (often simply called PageRank) (e.g., [6, 15, 49]). We can view the Internet as a directed graph, in which the websites are represented by vertices, and an edge (u, v) exists if there is a hyperlink on u that directs to v. The idea of PageRank is that important pages have many pages (including other important pages) linking to them, and unimportant pages have few, mostly unimportant pages linking to them. The importance of every vertex is a function of every other vertex. Because the Internet is huge, computing the exact PageRank of each vertex would take a very long time, and in addition, the Internet is constantly changing; we would like to be able to compute the approximate value of a website's PageRank by looking only at only a small number of other websites. Many of the PageRank algorithms developed (including the ones cited above) are in fact LCAs. Other examples of algorithms that are LCAs are locally decodable codes [51]: Suppose m is a string with encoding $y = E(m)$. On input x, which is close in Hamming distance to y, the goal of locally decodable coding algorithms is to provide quick access to the requested bits of m, by looking at a limited (sublinear) number of bits of x. More generally, the reconstruction models described in [2, 99] describe scenarios where a string that has a certain property, such as monotonicity, is assumed to be corrupted at a relatively small number of locations. Let P be the set of strings that have the property. The reconstruction algorithm gets as input a string x that is close (in Hamming distance) to some string y in P. For various types of properties of P, the above works construct algorithms that give fast query access to locations in y.

Part of what makes LCAs exciting is that they share the motivation and concepts of several major topics in theoretical computer science. This allows us to build upon

[1] What constitutes a "good" solution will vary from problem to problem, depending on our specific needs. For example, it is often enough to find an approximate solution.

well-known results and techniques and make new connections. Interestingly, as the connection is bidirectional, results that are discovered in research on LCAs are often immediately useful in other settings (for example, Even et al. [29] used LCAs to design distributed approximation algorithms to maximum matching). In the next subsections we give a history of LCAs along with the relationship to other work. We also highlight the contributions of this thesis.

1.1.1 Sublinear Approximation Algorithms

There has been much recent interest in devising local algorithms for problems on constant degree graphs and other sparse optimization problems. The goals of these algorithms have been to approximate quantities such as minimal vertex cover, maximal/maximum matching, and sparse packing and covering problems [55, 69, 79, 86, 89, 112]. See [23, 97] for surveys. One feature of these algorithms is that they show how to construct an oracle that, for each vertex (or edge), returns whether it is part of the solution whose size is being approximated - for example, whether it is in the vertex cover or maximal matching. Their results show that this oracle can be implemented in time independent of the size of the graph (depending only on the maximal degree and the approximation parameter). However, because their goal is only to compute an approximation of some quantity, they can afford to err on a small fraction of their local computations. Thus, their oracle implementations give LCAs for finding relaxed solutions to the the optimization problems that they are designed for.

Parnas and Ron [86] (see Chapter 3) gave a simple reduction from constant-time distributed labeling algorithms to sublinear approximation algorithms (a labeling algorithm is simply an algorithm that assigns vertices (or edges) labels that meet some set of constraints: a maximal independent set algorithm assigns each vertex a boolean label indicating whether or not it is in the independent set; a vertex coloring algorithm assigns each vertex a color).

Theorem 1.1.1. *[86] Let $G = (V, E)$ be a distributed network with degree at most d. Let $\mathcal{D}$ be a deterministic distributed algorithm that computes a labeling $D(G)$ in k rounds. Then it is possible, for any node $v \in V$ to compute $D(v)$ in time and space complexity $O(d^k)$ using a single processor, where the algorithm uses only neighbor and degree queries.*

Using this reduction, the approximation algorithm queries a constant number of

vertices, and based on the replies, computes an approximate size of the labeling problem. We can use the same reduction to obtain an LCA for labeling problems, on graphs of bounded degree, as the reply to every query will be consistent with the output of the distributed algorithm.

Rubinfeld et al. [98] designed LCAs for several problems such as hypergraph 2-coloring and MIS on graphs of bounded degree.[2] Their LCA for MIS is based on the Parnas-Ron reduction of Luby's distributed algorithm [65], only instead of running Luby's algorithm for $O(\log n)$ rounds (which would imply a $d^{O(\log n)}$ running time), it is stopped after a constant number of rounds. This guarantees that most of the vertices are accounted for (either in the independent set or neighbors of vertices that are in the independent set). They then used the Lovász Local Lemma to show that after the above vertices[3] are removed from the graph, any remaining connected component must be of size $O(\log n)$ with high probability (w.h.p.), and so the greedy algorithm can be used. Their focus was on the time bounds of these algorithms.

1.1.2 Simulating Online Algorithms

Nguyen and Onak [79] showed how to simulate certain online algorithms to obtain sublinear approximation algorithms for problems like vertex cover and maximum matching on graphs whose degree is bounded by a constant, by generating a random order on the vertices, and simulating some online algorithm $\mathcal{A}$ using this order. Their key observation is that in expectation, we only need to make a constant number of probes to the graph in order to determine the output of $\mathcal{A}$ on any vertex. Then, we can query a constant number of vertices to determine whether they are in, say, the vertex cover, to get an approximation to the size of the vertex cover. (If we find that we need too many probes for determining the reply to a specific vertex, we abort and query another vertex instead.) The idea behind the complexity analysis is the following: give each vertex, uniformly at random, a real number between 0 and 1, called its *rank*. Simulate the online algorithm on the order implied by these ranks (where the rank represents arrival time). Say we want to evaluate $\mathcal{A}$ on some vertex v. Create a tree rooted at v, and add all the neighbors of v that arrived before v. Continue to iteratively add neighbors of vertices in the tree until no more can be added. This *query tree* is an upper bound to

[2] The graphs need to obey certain other combinatorial properties, specifically, to meet the requirements of the Lovász Local Lemma.

[3] i.e., the vertices that are accounted for

the number of probes to the graph that we need to make in order to evaluate $\mathcal{A}$ on v. For intuition to why a query tree has constant size in expectation, assume that v has rank 0.5. In expectation, half of its neighbors arrive before it. Half of them will have a rank of at most 0.25, and most of their neighbors will arrive after them. Thus, the degrees of the vertices of the tree quickly drop. We would like to use the same reduction (generating a random order and simulating an online algorithm on this order) be used to obtain LCAs and not just approximation algorithms; the main difference between the two is that sublinear approximation algorithms are content with having a reply to a constant fraction of the queries, while LCAs require an answer for *every* query. Thus, an LCA cannot just abort if a computation takes too long. Fortunately, it turns out that w.h.p., the query tree will not be too big.

Our contributions. In [4], we bounded the size of the query tree to show that every query requires a polylogarithmic[4] number of probes with high probability (w.h.p.). The idea behind the proof is the following: assume that the maximal degree of the graph is d. Consider a partition of $[0,1]$ into $d+1$ equal intervals. We say that the vertices whose rank is in the interval $\left[\frac{i}{d+1}, \frac{i+1}{d+1}\right)$ are on *level* i. In the query tree, if there are k vertices on level i, it is unlikely that there will be much more than $O(k \log n)$ vertices on level $i+1$, as each vertex on level i has, in expectation, less than 1 neighbor on level $i+1$. In [67], we used a more subtle bounding of the query tree to show that the number of probes per query is logarithmic in the number of vertices w.h.p. In [90], we used a different technique to logarithmically bound the number of probes required by this simulation; the result holds for a larger family of graphs, which we discuss in Subsection 1.1.4. As the result [90] subsumes the results of [4] and [67], it is the one that we present in this dissertation, in Chapter 4.

1.1.3 Pseudorandom Orderings

Bounding the size of the query tree allows us to bound the number of probes to the graph, and hence the running time of the LCA (the running time is not exactly the number of probes, but they are usually closely related - see Chapter 4 for a more detailed discussion). These bounds depend on the fact that we use some randomness to determine the order; as we want the replies of the LCA to be consistent with the

[4] Specifically $O(\log^{d+1} n)$, where n is the number of vertices and d is the maximal degree of the graph.

same solution, we must use the same randomness every time. Therefore, we need to store this randomness, which affects our space requirements. If we wanted to store a permutation of the vertices, this would require $\Omega(n \log n)$ bits, assuming there are n vertices. Consider the most intuitive way to store a permutation: map each vertex v to a unique value in $\{1, 2, \ldots, n\}$; denote this mapping by π. Vertex v appears before vertex u in the permutation iff $\pi(v) < \pi(u)$. In order to store this mapping, we need to store $\log n$ bits for each vertex, for a total of $n \log n$ bits. Unfortunately, this is the best we can do asymptotically (see, e.g., [12]). We therefore use *pseudorandom generators*. A pseudorandom generator is an algorithm that takes as input a short, perfectly random seed and then returns a (much longer) sequence of bits that "looks" random. We clearly sacrifice some randomness when we do this: the permutation we get is defined by the random seed, and if the seed is of length k, there are only 2^k possible distinct values of the seed. However, in certain cases, this randomness can be good enough for our purposes. Consider the case that we want N random variables, but we don't need them to be completely independent; it suffices that every two of them are independent. This is known as *pairwise independence* and is quite common in randomized algorithms; possibly the best known example is Luby's MIS algorithm [65].

In Luby's algorithm, each vertex chooses itself with some (small) probability, and if none of its neighbors is chosen, it adds itself to the independent set (IS). The choices of any vertex that is not v's neighbor are irrelevant to whether or not v is added to the IS. In fact, all we require is that the choices of every two neighboring vertices are independent - we require pairwise independence. Although we need to generate n random variables, we do not need n random bits, in fact we only need $O(\log n)$ random bits. A very simple example of a construction of pairwise independent random variables is the following [56].

Assume for simplicity that n is a prime number, and that every vertex v has a unique ID between 1 and n (we represent this ID by v as well). Further assume that we want to generate for each vertex a (not necessarily unique) random label between 1 and n, such that the labels are pairwise independent. We can sample uniformly at random two integers, a and b, from $\{1, 2, \ldots n\}$, and consider the hash function $h(v) = av + b$. We claim that $h(v)$ is uniformly distributed, and every two labels are independent. It is easy to see why this is true: if we know $h(v)$ for some vertex v, $h(u)$ is uniformly distributed for any u. Notice however, that if we know $h(u)$ and $h(v)$, we can determine

a and b, and hence completely know h (and therefore the evaluation of h on any other $w \in \{1, 2, \ldots n\}$). Therefore h is pairwise independent, but not k-wise independent for any $k > 2$. This notion of k-wise independent hash functions was introduced by Carter and Wegman [18]. They also introduced *almost* k-wise independent hash functions, which we will touch upon shortly. The notion of limited independence has been used in many aspects of computer science; for an excellent introduction to pseudorandomness and k-wise independence, see [106].

Back to our scenario, we know (from bounding the query tree) that we will not require more than $O(\log n)$ probes per query, and so we only require $O(\log n)$-independence for our bounds to hold. In fact, we don't need $\log n$-wise independence - *almost* $\log n$-wise independence is sufficient for our purposes. Informally, a hash function h is almost k-wise independent if the evaluation of h on a set of at most k variables "looks" like the uniform distribution. In other words, it is difficult to distinguish between k truly random bits and the evaluation of the hash function on any k variables. We provide a more formal discussion in Chapter 4. Naor and Naor [75] showed that using almost k-wise independence allows us to further reduce the seed length; Alon et al. [3] showed simpler constructions of almost k-wise independent hash functions.

Our contributions. In [4], we showed that to generate the randomness necessary for the LCAs that we obtain using the reduction to online algorithms, we can use a random seed of length $O(\log^3 n)$, using an implementation of the constructions of [3]. The idea is the following: generate for each vertex an ($O(\log n)$-wise independent) integer in $[1, n^4]$. The probability that there is any collision is at most $\frac{1}{n^2}$. If there is a collision, we cannot guarantee the number of probes or running time, but the probability of this happening is negligible. In [90], we showed that, for the same reduction, it is sufficient to give each vertex a number in $[1, L]$, where L is some constant. There will (w.h.p.) be collisions, but even using an arbitrary tie-breaking rule, the number of probes to the graph can be shown to still be $O(\log n)$ w.h.p., and hence, using a new construction of almost k-wise independent random variables of [72], they showed that a seed of length $O(\log n)$ suffices. The latter result is presented in Chapter 4.

1.1.4 d-light Graphs

Most of the research on LCAs (e.g., [4, 28, 60, 68, 98, 98]) focuses on graphs of bounded constant degree. A natural question to ask is "(Other than graphs of bounded degree), which graphs admit LCAs?" We take an important step forward in characterizing these graphs.

Our contributions. In [67] (and later in [42]), we considered graphs where the degree is distributed binomially (and its expectation is constant). We showed that the size of the query tree is bounded by $O(\log n)$ w.h.p. in this case as well. In [90], we expanded the family for graphs for which the online reduction holds to a family of graphs called *d-light* graphs. The high-level idea behind d-light graphs is the following: given that some subgraph has already been exposed, we would like the degree of the next exposed vertex to be bounded by a distribution that has expected value d and whose tail is light. A formal definition is given in Section 2.3. We will see that this definition encompasses a broad range of graphs, for example:

- Graphs with degree bounded by d, where $d = O(\log \log n)$,
- The random graphs $G(n, p)$, where $p = d/n$, for any $d = O(\log \log n)$,
- Bipartite graphs on n consumers and m producers, where each consumer is connected to d producers at random.

In Chapter 3, we show that the Parnas-Ron reduction can be used on d-light graphs to obtain an LCA whose number of probes and running time is $O(\log n)$ w.h.p.

1.1.5 Graphs of Constant Bounded Degree

The family of graphs of constant bounded degree has been extremely well studied in the context of distributed algorithms (in particular, the LOCAL model [87]). Naor and Stockmeyer [76] investigate the question of what can be computed on these networks in constant time, and show that there are nontrivial problems under these constraints. They investigate locally checkable labeling (LCL) problems, where the legality of a labeling can be checked in constant time (e.g., coloring). They describe constant-time algorithms for several problems, notably for c-weak coloring,[5] for some constant c, on

[5] c-weak coloring means coloring the vertices with at most c colors such that every vertex has at least one neighbor colored differently from itself

graphs of odd degree. Cole and Vishkin [21] showed that it is possible to obtain a 3-coloring of an n-cycle in $O(\log^* n)$ communication rounds (assuming that there are unique node identifiers). This was shown to be tight by Linial [61]. This field has received considerable interest over the past three decades, e.g., [38, 54, 55, 85]; see [87] for an introductory book and [100] for a recent (2013) survey.

Even et al. [28] investigate the connection between local distributed algorithms and LCAs. They show how to color the vertices in a small neighborhood of a graph to obtain an acyclic orientation of the neighborhood. Their coloring algorithm is an adaptation of the techniques of Panconesi and Rizzi [85] and Linial [61], and produces a coloring in $O(\log^* n)$ rounds. This orientation defines an order on the vertices, and an online algorithm can be simulated on this order, similarly to the technique presented in Subsection 1.1.2. This generates a deterministic LCA for MIS on constant degree graphs that requires $O(\log^* n)$ probes.

Our contributions. In [66], we showed that there are some non-trivial LCAs that only require a constant number of probes (and running time), removing the dependency on n completely. Specifically, they give an LCA that computes an approximately optimal maximal weight forest; LCAs for multicommodity flow and multicut on trees; and give a constant-time reduction from weighted matchings to unweighted matchings. These results are presented in Chapter 6.

1.2 Algorithmic Game Theory

Algorithmic game theory (AGT) is an area in the intersection of game theory and algorithm design, whose objective is to design algorithms in strategic environments: the input, or some part of it, is given to the algorithm by selfish agents that have a stake in the outcome. Therefore, unlike in classical algorithm design, we cannot assume that the input given to us is correct, and so we need to incentivize the agents to truthfully report their part of the input. The field was started by Nisan and Ronen in their seminal 1999 paper [81]. Since then, it has flourished, with thousands of papers being written on it. We cannot hope to provide even a brief summary of the many interesting results of AGT - we refer the reader to [80] as a good starting point. In this work we tackle three very different game-theoretic scenarios: stable matching (Chapter 8), machine scheduling (Chapter 9) and combinatorial auctions (Chapter 10).

In the stable matching problem, there is a set of men and a set of women, where each man has a preference list over the women and each woman has a preference list over the men. We want to find a matching that is *stable*; i.e., that there is no pair of a man and a woman who prefer each other over their given partner (and hence would prefer to defect from the suggested matching). In machine scheduling problems, we would like to assign jobs to machines so that the last job terminates as quickly as possible, and in combinatorial auctions, we would like to assign items to buyers so that the social welfare (or "total happiness") is maximized. In the last two settings, we use payments in order to ensure that the agents (the machines and buyers respectively), report truthfully (their computational power and how much they value the items respectively). Because the three settings are very different, we provide a brief introduction to each of them (along with a discussion of related work) in their respective chapters.

1.3 Overview of the Thesis

1.3.1 Part I - Designing Local Computation Algorithms

In the first part of the thesis, we introduce the model of local computation algorithms and describe some techniques for designing LCAs. In Chapter 2, we introduce the model and define the performance criteria for LCAs on graphs. The five criteria are: the number of probes the LCA makes to the graph per query; the memory required to store the source of randomness, if our LCA is randomized; the running time per query; the space requirement for computing the reply to a query; and the probability that the LCA deviates from the specified number of probes, time or space.

We then introduce a family of graphs, which we call *d-light* graphs: A d-light graph is a graph that is sampled from some distribution of graphs, such that whenever some subgraph has already been exposed, the degree of the next exposed vertex is bounded by a light-tailed distribution that has expected value d. Many natural graphs fall within this family, including graphs of bounded degree and random (Erdős-Rényi) graphs. We show that given that some sufficiently large subgraph has been exposed, then w.h.p. the neighborhood of the subgraph is not much bigger than the subgraph itself.

We end Chapter 2 by showing that some problems, even ones that are solvable in polynomial time, may not have LCAs. Specifically, we show that there can be no LCA for maximum matching.

In Chapters 3 and 4 we show how to we can transform certain distributed and online algorithms, respectively, to LCAs. The idea behind the reduction to distributed algorithms is as follows: If we are queried on a vertex v and there is a distributed algorithm $\mathcal{D}$ for the problem terminates after a constant number of rounds k, then we can simulate $\mathcal{D}$ on all vertices at distance at most k from v. If the graph is d-light, we show that the total number of these vertices is at most logarithmic in the size of the graph, w.h.p. The reduction to online algorithms is conceptually (almost) as simple: we generate some random order on the vertices and simulate an online algorithm $\mathcal{A}$ on this order. Because we want the replies to all queries to be consistent with a single feasible solution, we have to simulate $\mathcal{A}$ on the same order for every query. We therefore have to remember this ordering. Unfortunately, our results depend on this order being random, and so we cannot use some arbitrary ordering (for example, ordering the vertices by ID). Although we require that the ordering is random, we do not require complete randomness. It turns out that a logarithmic number of truly random bits suffices for our purposes. Using this small seed, we can generate an "almost" random order on the vertices that still guarantees that we only probe the graph a logarithmic number of times per query in the worst case, w.h.p. We also show that in expectation, we only require a constant number of probes. In Chapter 5, we show an LCA for approximating the maximum matching on graphs of constant degree.

We end the first part of the thesis by showing that we can sometimes design LCAs whose running time is independent of the size of the graph, and depends only on the maximal degree (Chapter 6). We design and LCA for finding a forest whose weight is a $(1-\epsilon)$ approximation to the weight of the maximal spanning forest, and LCAs for computing integer multicommodity flow and multicuts on trees that are a 4- and $\frac{1}{4}$-approximation to the maximal flow and minimal cut respectively. Finally, we show that, given a constant-time α-approximation algorithm ($\alpha < 1$) for unweighted matching, we can design a constant-time $\frac{\alpha}{8}$-approximation algorithm for weighted matching. The running time of our weighted matching LCA is independent of the weight function as well as the size of the graph.

1.3.2 Part II - Local Computation Mechanism Design

In the second part of the thesis, we design *local computation mechanisms* (LCMs) - game-theoretic mechanisms that run in polylogarithmic time and space. In Chapter 7

we formally define our model and give a simple example - an implementation of the *random serial dictatorship* algorithm. In the *housing allocation problem*, there is a list of houses and a list of buyers, and the buyers have a preference list over the houses. We would like to find a "good" allocation of houses to buyers. The random serial dictatorship algorithm has some nice game-theoretic properties, such as the buyers having no incentive to lie about their preference list. The local mechanism requires that the input obey some restrictions (specifically that each buyer's preference list includes a constant number of houses, chosen uniformly at random), and, when queried about a buyer, returns which house she is allocated, while still guaranteeing all the nice properties that the non-local mechanism has.

In Chapter 8, we show how to implement the Gale-Shapley stable matching algorithm [34] as an LCA in the setting where each man's preference list has a constant number of women, selected uniformly at random. In the Gale-Shapley algorithm, each man approaches his most preferred woman. The woman tentatively accepts the man she prefers out of all of the men that approached her, and rejects the rest. In the next round, every man that was just rejected approaches the next woman on his list. Each woman once again tentatively accepts the man whom she prefers (out of all the men standing in front of her, including the man she tentatively accepted in the previous round, if there was one). This continues until each man is standing in front of a single woman, at which point the algorithm terminates. Gale and Shapley showed that this matching is stable. Our local implementation is straightforward: we simulate this algorithm for a constant number of rounds, k, and disqualify any man that is rejected on round k. We show that by appropriately selecting the number of rounds, we can guarantee that at most an ϵ fraction of the men is disqualified, for any ϵ. This assures that our matching is ϵ-almost stable (at most an ϵ fraction of the men (and women) would prefer each other to their matched partner). We use our techniques to show that in the general case when the men's lists have bounded length (even in cases that do not admit an LCA), we can find arbitrarily good matchings[6] (up to both additive and multiplicative constants) by truncating the Gale-Shapley algorithm to a constant number of rounds.

In Chapter 9, we consider the problem of scheduling jobs on machines. In the *makespan minimization* problem, we want to schedule n jobs on m machines so as

[6] See Section 8.1 for a formal discussion.

to minimize the maximal running time (makespan) of the machines. This problem has many variations; we consider the scenario in which m identical jobs need to be allocated among n related machines. The machines are strategic agents, whose private information is their speed. We show:

1. A local mechanism that is truthful in expectation[7] for scheduling on related machines, that provides an $O(\log \log n)$-approximation to the optimal makespan.

2. A local mechanism that is universally truthful[8] for the restricted case (i.e., when each job can run on one of at most a constant number of predetermined machines), that provides an $O(\log \log n)$-approximation to the optimal makespan.

We also show some subtle and surprising results on the truthfulness of our algorithms.

In Chapter 10, we consider matching combinatorial auctions. *Combinatorial auctions* (CAs) are auctions in which buyers can bid on bundles of items. We consider the following scenario: m items are to be auctioned off to n unit-demand buyers, where each buyer is interested in a set of at most k items, sampled uniformly at random. We show universally truthful local mechanisms for the following variations, both of which give a $\frac{1}{2}$-approximation to the optimal solution:

1. When all buyers have an identical valuation for the items in their sets, and the buyers' private information is the sets of items they are interested in.

2. When the sets are public knowledge, and the buyers' private information is their valuation.

If each buyer is interested in a set of at most k items, and has private valuation for this set, we show a universally truthful local mechanism that admits a $\frac{1}{k}$-approximation to the optimal social welfare.

1.4 Published and Submitted Papers

- Chapter 4 is based on

[7] For formal definitions of truthfulness in expectation and universal truthfulness, see Section 9.1.
[8] See Footnote 7.

  - [4]: *Space-Efficient Local Computation Algorithms.* Noga Alon, Ronitt Rubinfeld, Shai Vardi and Ning Xie. In Proc. 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 1132–1139, 2012,
  - [67]: *Converting Online Algorithms to Local Computation Algorithms.* Yishay Mansour, Aviad Rubinstein, Shai Vardi and Ning Xie. In Proc. 39th International Colloquium on Automata, Languages and Programming (ICALP), pages 653–664, 2012,
  - [90]: *New Techniques and Tighter Bounds for Local Computation Algorithms.* Omer Reingold and Shai Vardi. In Journal of Computation and System Sciences (JCSS), 82(7), pages 1180–1200, 2016.

- Chapter 5 is based on [68]: *A Local Computation Approximation Scheme to Maximum Matching.* Yishay Mansour and Shai Vardi. In APPROX–RANDOM, pages 260–273, 2013.

- Chapter 6 is based on [66]: *Constant-Time Local Computation Algorithms.* Yishay Mansour, Boaz Patt-Shamir and Shai Vardi. In 13th Workshop on Approximation and Online Algorithms (WAOA), pages 110–121, 2015.

- Part II is based on [42]: *Local Computation Mechanism Design.* Avinatan Hassidim, Yishay Mansour and Shai Vardi. In ACM Conference on Economics and Computation, EC '14, pages 601-616, 2014. To appear in *Transactions on Economics and Computation.*

Part I

Designing Local Computation Algorithms

Chapter 2

The Local Computation Model

In this chapter we formally define the model of local computation algorithms. We first introduce some notation.

2.1 Notational Conventions

We denote the set of integers $\{1, 2, \ldots, n\}$ by $[n]$. We write $\mathbb{N}$ for the set of nonnegative integers, and $\mathbb{R}$ for the set of reals.

Let X be a random variable, distributed according to the Binomial distribution with parameters n and p; we denote this by $X \sim B(n, p)$.

Let $G = (V, E)$ be a simple undirected graph. The neighborhood of a vertex v, denoted $N(v)$, is the set of vertices that share an edge with v: $N(v) = \{u : (u, v) \in E\}$. The *degree* of a vertex v, is $|N(v)|$. The neighborhood of a set of vertices $S \subseteq V$, denoted $N(S)$, is the set of all vertices $\{u : v \in S, u \in N(v) \setminus S\}$. The *distance* between two vertices u and v, denoted $\mathrm{dist}(u, v)$, is the minimal number of edges required to reach v from u (or vice-versa). For any vertex v, its k-neighborhood (for $k \geq 0$), denoted $N^k(v)$, is the set of all vertices at distance at most k from v. For any edge $e = (u, v)$, its k-neighborhood is defined as $N^k(e) = N^k(u) \cup N^k(v)$.

We assume that each vertex $v \in V$ has a unique label between 1 and some $n = \mathrm{poly}(|V|)$. For simplicity, assume $n = |V|$ (or in other words $V = [n]$).

We use "with high probability" (w.h.p.) to mean with probability at least $1 - \frac{1}{\mathrm{poly}(n)}$, where $\mathrm{poly}\, n$ is some polynomial in n. Although we state our results with a specific (asymptotic) failure probability, such as $\frac{1}{n^2}$, they can all easily be extended to have a failure probability of $\frac{1}{n^c}$, for any constant c.

2.2 The Model

We assume the standard uniform-cost RAM model, in which the word size is $O(\log n)$ bits, where n is the input size, and it takes $O(1)$ to read and perform simple words operations.

Before formally defining LCAs, we define the measures by which the complexity of LCAs is quantified.

- *Number of probes.* An LCA can access a vertex in the input graph and ask for a list of its neighbors, or the ID of a neighbor. This is called a "probe".
- *Running time.* The running time of an LCA, per query, is the number of word operations that the LCA is required to perform in order to output a reply to the query. In the running time calculation, a probe to the graph is assumed to take $O(1)$. Note, however, that if we wish to perform an operation on a the entire list of neighbors and the list is of size ℓ, (such as committing the list to memory or finding the maximal ID), this will take $O(\ell)$ operations.
- *Enduring memory.* As a preprocessing step, before it is given its first query, the LCA can be allocated some memory to which it is allowed to write. Once the first query is given to the algorithm, it can only read from this memory and can never modify it. This can be viewed as the LCA being allowed to augment the input with some small number of bits.
- *Transient memory.* This is simply the amount of memory (measured in the number of words) that an LCA requires in order to reply to a query. It does not include the enduring memory.
- *Failure probability.* The LCA is allowed to deviate from the number of probes, running time or transient memory that it requires per query. In this case, we say that the algorithm "fails". We require that, even if the LCA is queried on all vertices, the probability that it will fail on any of them is still very low. Note that the algorithm is never allowed to reply incorrectly - the failure is only a function of its complexity. Note further that the LCA is never allowed to use more enduring memory than it requested.

We define LCAs as follows.

Definition 2.2.1 (Local computation algorithm)**.** *A* $(p(n), t(n), em(n), tm(n), \delta(n))$-local computation algorithm $\mathcal{A}$ *for a computational problem is a (possibly randomized) algorithm that receives an input of size* n*, and a query* x*. Before the first query,* $\mathcal{A}$ *is allowed to write* $em(n)$ *bits to the enduring memory, and may only read from it thereafter. Algorithm* $\mathcal{A}$ *makes at most* $p(n)$ *probes to the input in order to reply to any query* x*, and does so in time* $t(n)$ *using* $tm(n)$ *transient memory (in addition to the enduring memory). The probability that* $\mathcal{A}$ *deviates from the probe, time or transient memory bounds is at most* $\delta(n)$*, which is called* $\mathcal{A}$*'s failure probability. Algorithm* $\mathcal{A}$ *must be* consistent*, that is, the algorithm's replies to all of the possible queries combine to a single feasible solution to the problem.*

2.3 d-light Graphs

In this section we introduce the family of graphs for which our reductions to distributed and online algorithms apply. We then prove a probabilistic bound on the size of the neighborhood of any set of adaptively exposed vertices of such a graph. We only discuss *undirected graphs,* but both the definitions and algorithms easily extend to the directed case. We call these graphs *d-light*; they generalize a large family of graphs (see below). Before defining d-light graphs, we need to define certain processes that adaptively expose vertices of a graph. A *vertex exposure procedure* is a procedure $\mathcal{P}$ that is allowed to probe vertices in the graph (usually according to some predetermined set of rules), to obtain their list of neighbors. Such procedures can be used, for example, to adaptively learn the structures of neighborhoods of graphs. Once a vertex has been probed by $\mathcal{P}$, it is said to be *exposed*.

Definition 2.3.1 (Adaptive vertex exposure)**.** *An* adaptive vertex exposure process $\mathcal{P}$ *is a process that receives a limited oracle access to a graph* $G = (V, E)$ *in the following sense:* $\mathcal{P}$ *maintains a set of vertices* $S \subseteq V$*, (initially* $S = \emptyset$*), and updates* S *iteratively. In the first iteration,* $\mathcal{P}$ *exposes an arbitrary vertex* $v \in V$*, learns the IDs of all of the neighbors of* v*, and adds* v *to* S*. In each subsequent iteration,* $\mathcal{P}$ *can choose to expose any* $u \in N(S)$*:* u *is added to* S*, and* $\mathcal{P}$ *learns the IDs of* u*'s neighbors. If a subset* $S \subseteq V$ *was exposed by such a process, we say that* S *was* adaptively exposed*.*

Before defining d-light graphs, we need one more definition.

Definition 2.3.2 (Stochastic dominance)**.** *For any two distributions over the reals, X and Y, we say that X is* stochastically dominated by[1] *Y if for every real number x it holds that $\Pr[X > x] \leq \Pr[Y > x]$. We denote this by $X \leq_{st} Y$.*

For a more in-depth discussion of the properties of (and some new results on) stochastic dominance that we will require, see Appendix A.2.

Definition 2.3.3 (s-conditional d-light distribution)**.** *Let $d, s > 0$, $G = (V, E)$ be a graph sampled from some distribution, and $\mathcal{P}$ be an adaptive vertex exposure process. If, for every $S \subset V$, $|S| \leq s$, that is adaptively exposed by $\mathcal{P}$, conditioned on any instantiation of $S \cup N(S)$ and set of edges $E_S = \{(u, v) \in E \mid u \in S\}$, we have that for every $v \in N(S) \setminus S$ (or any v in the case that S is empty), there is a value $d \leq \alpha(v, S) \leq |V|$, such that $|N(v) \setminus S|$ is stochastically dominated by $B\left(\alpha(v, S), \frac{d}{\alpha(v,S)}\right)$, we say that the degree distribution of G is s-conditionally d-*light *(or, for simplicity, that G is s-conditionally d-light). If this property holds for $s = |V| - 1$, we say that G is d-light.*

Remark 2.3.4. *We shall see that, because we are interested in LCAs whose complexity measures are polylogarithmic, it suffices to consider* polylog n*-conditionally d-light graphs. For simplicity, we state most of our results in terms of d-light graphs. All of these results also hold for* polylog n*-conditionally d-light graphs; the proofs may need to be slightly modified, and we remark on these differences when they arise. We especially note that the results of Chapter 4 hold for* (polylog n)*-conditionally d-light graphs without modification.*

The family of d-light and conditionally d-light graphs includes many well-studied graphs in the literature; for example,

- d-regular graphs, or more generally, graphs with degree bounded by d (taking $\alpha = d$),
- The random graphs $G(n, p)$, where $p = \frac{d}{n-1}$. Each one of the $\binom{n}{2}$ edges is selected independently with probability p; therefore the degree of each vertex is distributed according to the binomial distribution $B(n - 1, \frac{d}{n-1})$. Note that in general, the degrees are not independent (for example, if one vertex has degree $n - 1$ then all

[1] This is usually called *first-order stochastic dominance*. As this is the only measure of stochastic dominance we use, we omit the term "first-order" for brevity.

other vertices are connected to this vertex). However, once a subset $W \subset V$ has been exposed (as well as the edges in the cut $(W, G \setminus W)$, for any $v \notin W$, $|N(v) \setminus W| \sim B(n - |W| - 1, \frac{d}{n-1})$, which is stochastically dominated by $B(n-1, \frac{d}{n-1})$.

- Bipartite graphs on n consumers and n producers, where each consumer is connected to d random producers are $(n/2)$-conditionally $2d$-light. See Claim 2.3.5.

Claim 2.3.5. *The family of bipartite graphs on n consumers and n producers, where each consumer is connected to $d \ll n$ producers uniformly at random is $(n/2)$-conditionally $2d$-light.*

Proof. Let S be any adaptively exposed set of size at most $n/2$. We prove that the degree of any producer/consumer $i \notin S$ is stochastically dominated by $B\left(\alpha(i,S), \frac{2d}{\alpha(i,S)}\right)$. (This is stronger than the requirements of Definition 2.3.3 in that (1) we prove it for every vertex in $V \setminus S$ which is a superset of $N(S) \setminus S$, and (2) we bound $|N(v)|$, and $|N(v)| \geq |N(v) \setminus S$.)

The degree of any consumer i (in S or not) is at most d - which is trivially stochastically dominated by $B(2d, 1)$ (taking $\alpha(i, S) = 2d$). The degree of any producer i not in S is stochastically dominated by $B\left(n, \frac{d}{n/2}\right) = B(n, 2d/n)$: Let $P \subseteq S$ denote the set of producers in S. Set $|P| = q \leq n/2$. Consider a consumer j, and let r be the number of neighbors it has in P. j chooses i w.p. $\frac{d-r}{n-q} \leq \frac{2d}{n}$. As there are at most n consumers, each choosing i w.p. at most $2d/n$, taking $\alpha(i, S) = n$ completes the proof of the claim. □

Assumption 2.3.6. *When considering d-light graphs (or conditionally d-light graphs) in this work, we assume that d is a constant.*

In fact, as we show in Section 4.5, we can allow d to be as large as $\Theta(\log\log n)$. However, we only explicitly compute the complexity of the LCAs on d-light graphs for constant d. If d is super-constant, the complexity parameters deteriorate. We discuss this further in Section 4.5, but for now we state that for the LCAs on d-light graphs in this work, the number of probes, running time, enduring memory and transient memory all remain polylogarithmic for $d = O(\log\log n)$.

2.3.1 Bounding the Neighborhood of Exposed Sets

Many of the results on d-light graphs of this thesis depend on a crucial property: any exposed subgraph of a d-light graph does not have "too many" neighbors. For

motivation, consider the following strong property:

Property 2.3.7. *Every connected subgraph with s vertices has at most $O(d \cdot s)$ neighbors.*

Property 2.3.7 holds trivially for graphs of degree bounded by d, but it does not necessarily hold for d-light graphs. We could ask for a weaker property:

Property 2.3.8. *Every connected subgraph with s vertices has at most $O(d \cdot s)$ neighbors w.h.p.*

Unfortunately, Property 2.3.8 does not hold for d-light graphs either. It applies to large subgraphs, but not small ones. Consider the subgraph that is a single vertex, whose degree is distributed binomially $deg(v) \sim B(n, d/n)$. With probability $\Omega(1/n)$, $deg(v) = \Omega(\frac{\log n}{\log \log n})$. We therefore ask for two weaker properties, given by Properties 2.3.10 and 2.3.11. The first property is that when we adaptively expose a large enough connected subgraph, the number of neighbors it has is unlikely to be much larger than the subgraph itself. The second property is that if we adaptively expose a small enough connected subgraph, the number of neighbors it has is unlikely to be more than $c \log n$, for some constant c. The proofs of both properties depend on the following proposition. Note that in the following, we are counting the number of edges, one of whose endpoints (at least) is in S. The other endpoint may or may not be in S.

Proposition 2.3.9. *Let $\{X_{i,j}\}_{i \in [m], j \in [n^2]}$ be $n^2 \cdot m$ independent Bernoulli random variables such that $Pr[X_{i,j} = 1] = 2d/n^2$ for every $i \in [m]$ and $j \in [n^2]$. For every (s-conditionally) d-light graph $G = (V, E)$ with $|V| = n$ and every adaptively exposed subset of the vertices $S \subseteq V$ of size m (where $m \leq s$),*

$$|\{(u, v) \in E \mid u \in S\}| \leq_{st} 2dm + \sum_{i=1}^{m} \sum_{j=1}^{n^2} X_{i,j}.$$

Proof. Denote $|S| = m$. Label the vertices of S: $1, 2, \ldots m$, according to the order of exposure. That is, vertex 1 is the first vertex that was exposed, and so on. Denote by S_i the set of vertices $\{1, 2, \ldots, i\}$, and by Y_i the random variable representing the number of neighbors of vertex i that are not in S_{i-1}. That is, $Y_i = |N(S_i) \setminus S_{i-1}|$. The quantity we would like to bound, $|\{(u, v) \in E \mid u \in S\}|$, is at most $\sum_{i=1}^{m} Y_i$.

From the definition of s-conditional d-light distributions, conditioned on every possible realization of S_{i-1}, $N(S_{i-1})$ and the edges adjacent to S_{i-1}, there exists some

$d \leq \alpha_i \leq n$ such that $Y_i \leq_{st} B(\alpha_i, d/\alpha_i)$. By Lemma A.2.8, under the same conditioning, $Y_i \leq_{st} Z \sim 2d + B(n^2, \frac{2d}{n^2})$. This implies that conditioned on any realization of $Y_1, \ldots, Y_{i-1}$ we have that $Y_i \leq_{st} Z$. By Lemma A.2.3 we now have that $\sum_{i=1}^{m} Y_i$ is stochastically dominated by the sum of m independent copies of Z.

Note that for every i we have that $\sum_{j=1}^{n^2} X_{i,j} \sim B(n^2, 2d/n^2)$. By the definition of Z we now have that

$$\sum_{i=1}^{m} Y_i \leq_{st} 2dm + \sum_{i=1}^{m} \sum_{j=1}^{n^2} X_{i,j}.$$

□

Property 2.3.10. *There exists some constant $c > 0$, such that for every (s-conditionally) d-light graph $G = (V, E)$ with $|V| = n$ and every adaptively exposed subset of the vertices $S \subseteq V$ of size at least $c \log n$ (and at most s), we have that* $\Pr[|\{(u, v) \in E \mid u \in S\}| > 6d|S|] \leq 1/n^5$. *In particular,* $\Pr[|N(S)| > (6d-1)|S|] \leq 1/n^5$.

Proof. Letting $|S| = m$, from Proposition 2.3.9,

$$|\{(u, v) \in E \mid u \in S\}| \leq_{st} 2dm + \sum_{i=1}^{m} \sum_{j=1}^{n^2} X_{i,j}.$$

By the linearity of expectation, $\mathbb{E}[\sum_{i=1}^{m} \sum_{j=1}^{n^2} X_{i,j}] = 2dm$. By the Chernoff bound, there exists a constant c such that when $m \geq c \log n$ it holds that $\Pr[\sum_{i=1}^{m} \sum_{j=1}^{n^2} X_{i,j} > 4dm] \leq 1/n^5$. □

Property 2.3.11. *There exist constants $c'' > c' > 0$, such that for every d-light graph $G = (V, E)$ with $|V| = n$ and every adaptively exposed subset of the vertices $S \subseteq V$ of size at most $c' \log n$, we have that* $\Pr[|\{(u, v) \in E \mid u \in S\}| > c'' \log n] \leq 1/n^5$. *In particular,* $\Pr[|N(S)| > c'' \log n] \leq 1/n^5$.

Proof. Similarly to the proof of Corollary 2.3.10, let$|S| = m$. From Proposition 2.3.9, $|\{(u, v) \in E \mid u \in S, v \notin S\}| \leq_{st} 2dm + \sum_{i=1}^{m} \sum_{j=1}^{n^2} X_{i,j}$. By the linearity of expectation, $\mathbb{E}[\sum_{i=1}^{m} \sum_{j=1}^{n^2} X_{i,j}] \leq 2dc' \log n$. By the Chernoff bound of Theorem A.1.1, taking $c'' > 4edc'$,

it holds that $\Pr[\sum_{i=1}^{m}\sum_{j=1}^{n^2} X_{i,j} > c'' \log n] \leq 1/n^5$. □

2.4 Crisp Algorithms

We provide general reductions of LCAs to distributed and online algorithms, and we wish to bound the complexity measures of our LCAs. The number of probes is independent of the implementation of the online algorithm, and as we shall see later on, the amount of enduring memory required will usually not be affected, as long as the implementation is "reasonable". However the running time and space requirements clearly depend on the running time and space requirements of the simulated algorithm: If we probe one vertex, but the simulated algorithm performs some computation that requires time proportional to the number of bits used to represent the vertex's ID, the running time of the LCA will be $O(\log n)$, and not $O(1)$. However, this is not usually the case with online algorithms for combinatorial problems, and many algorithms "behave well". To quantify what we mean by this, we introduce the following definition.

Definition 2.4.1. *We say that an algorithm $\mathcal{A}$ is* crisp *if $\mathcal{A}$ requires time and space linear in its input length (where the input length is measured by the number of words), and the output of $\mathcal{A}$ is $O(1)$ per query.*

Most of the algorithms that we wish to convert to LCAs using the techniques of this paper are indeed crisp; for example the greedy algorithm for maximal independent set requires computation time and space linear in the number of neighbors of each vertex, and the output per vertex is a single bit. Not all the algorithms we wish to handle are necessarily crisp; in the case of vertex coloring, the output of the greedy algorithm can be $\log \Delta$, where Δ is the maximal degree of the graph. Nevertheless, as to avoid a cumbersome statement of our results, we restrict ourselves to crisp algorithms; it is straightforward to extend our results to non-crisp algorithms.

We note that the analysis of distributed algorithms often assumes that the "expensive" part of the computation is the message passing, while the processors themselves have unrestricted computational power. Nevertheless, many distributed algorithms are actually very efficient.

2.5 An Impossibility Result for Maximum Matching

In the rest of the thesis, we will focus on designing LCAs for problems. We remark that not all problems that are polynomial-time tractable admit LCAs. An example of one such problem is maximum matching (either weighted or unweighted). Efficient polynomial algorithms exist, such as Edmonds' celebrated blossom algorithm [26] but unfortunately, no LCA can exist. In fact, no LCA can exist for the simpler problem of finding a maximum matching on bipartite graphs. This is shown by the theorem and corollary below.

Theorem 2.5.1. *There does not exist an LCA for the maximum matching problem.*

Proof. To see why it is not always possible to solve the maximum matching locally, consider the following family of homomorphic graphs: $\mathcal{G} = \{G_i\}$. All $G_i \in \mathcal{G}$ have $2n$ vertices: $\{v_1, v_2, \ldots, v_{2n}\}$. In each $G_i = (V_i, E_i)$, vertices $V_i \setminus \{v_i\}$ comprise an (odd) cycle, and vertex v_i is connected to vertex v_{i-1} (modulo $2n$). Each G_i has a unique maximum matching. We are given as input a graph $G \in \mathcal{G}$, (i.e., we know it is G_i for some i, but we don't know the value of i). We would like to know whether the edge $e = (v_1, v_2)$ is in the maximum matching. Note that the edge e will be in exactly half of the maximum matchings.

Assume w.l.o.g. that the graph is either G_n or G_{n+1}. In the distributed model, this implies that the distance between the edge $e = (v_1, v_2)$ and the vertices that distinguish between of G_n and G_{n+1} (vertex v_{n-1} and vertex v_n) is $n - 2$ edges, which will be a lower bound on the time to detect the correct graph.

In the local computation model, we can write the edges in a random order. This implies that one needs to query, on average, $n-2$ edges to distinguish between G_n and G_{n+1}.

Therefore there cannot exist an LCA for maximum matching. □

Corollary 2.5.2. *There does not exist an LCA for the maximum matching problem in bipartite graphs.*

Proof. The proof is similar to the general case. In the bipartite case, though, in each G_i, the vertices $v_1, v_2, \ldots, v_{i-1}, v_{i+2}, \ldots, v_{2n}$ comprise an *even* cycle. Vertex v_i is connected to vertex v_{i-1} and vertex v_{i+1} is connected to vertex v_{i+2} (modulo $2n$). Note that the edges (v_{i-1}, v_i) and (v_{i+1}, v_{i+2}) must be in the maximum matching, because the

maximum matching in this case is of size n, meaning all vertices must be matched, including v_i and v_{i+1}. □

Chapter 3

Reducing LCAs to Distributed Algorithms

The next few chapters describe general techniques for designing LCAs. We build upon the vast literature on distributed and online algorithms, and give black-box reductions of LCAs to distributed and online algorithms for certain families of problems and graphs. In this chapter, we describe a simple method for converting any distributed algorithm that requires a constant number of rounds to an LCA. This reduction was proposed by Parnas and Ron [86]. After describing the reduction, we show how to improve the trivial bounds on the number of probes, running time and memory requirements when the resulting LCA is executed on d-light graphs.

3.1 The Parnas-Ron Reduction

Let $G = (V, E)$ be a graph, and let O be some arbitrary finite set. Parnas and Ron [86] noticed the following: Assume that there is a (crisp) distributed algorithm $\mathcal{A}$ that computes some function $F : V \to O$. If $\mathcal{A}$ terminates after a constant number of rounds, ℓ, then it is suffices to simulate $\mathcal{A}$ on the ℓ-neighborhood of any vertex v to obtain the value of $F(v)$ (Theorem 1.1.1). It is easy to see why this is true: if $\mathcal{A}$ terminates after ℓ rounds, no information from a vertex whose distance from v is greater than ℓ can reach v. In fact, this reasoning shows that it suffices to simulate the first round of $\mathcal{A}$ on the $N_\ell(v)$, the second round on $N_{\ell-1}(v)$, and so on. For simplicity, we assume that the distributed algorithm is simulated on the entire neighborhood for ℓ rounds, as this

Algorithm 1: Distributed Maximal Matching Algorithm for Bipartite Graphs

Input : $G = (U \cup W, E)$ with degree bounded by d
Output: A matching M
`// All ties are broken by ID/port number`

$M \leftarrow \emptyset$;
for $i \leftarrow 1$ *to* d **do**
  All unmatched $u \in U$ send a proposal to the first available neighbor;
  Each $w \in W$ accepts the first proposal it receives;

does not asymptotically affect the running time.[1] This reduction immediately allows us to convert any ℓ-time distributed algorithm to an $(O(d^\ell), O(\ell d^\ell), 0, O(\ell d^\ell), 0)$-LCA for graphs of degree bounded by d.

As an example, let us consider the simple distributed maximal matching algorithm for bipartite graphs of Hanckowiak et al. [41], Algorithm 1. Let $G = (U \cup W, E)$ be a bipartite graph. In each round, each vertex $u \in U$ sends a proposal to one of its unmatched neighbors, and each vertex $w \in W$ that was proposed to, accepts one of the proposals. If the maximal degree is d, it is easy to see that after d rounds, we have a maximal matching: Denote the matching found by the algorithm after d rounds by $\mathcal{M}$. Consider any edge $e = (u, w)$. If e was considered at some round by u, then w must be matched, either to u or to another vertex, and hence $\mathcal{M} \cup \{e\}$ is not a matching. If e was not considered, this is because either u or w was already matched. (Note that u will never propose to any neighbor more than once.) Using Theorem 1.1.1, we immediately get a $(O(d^d), O(d^d), 0, O(d^d), 0)$-LCA for this problem, on graphs of degree bounded by d.

Consider now the following scenario: there is a set M of n men and a set W of n women, and each man chooses d women uniformly at random. This scenario is quite common - it can be found in the context of, e.g., stable matchings [47, 53], see also Chapter 8; and load balancing [9, 14, 108], see also Chapter 9. (In the case of load balancing, M would be a set of balls/jobs and W the set of bins/machines). By Claim 2.3.5, this graph is $(n/2)$-conditionally $2d$-light; applying the Parnas-Ron reduction to s-conditionally d-light graphs guarantees that the number of probes, running time and transient memory are $O(\log^\ell n)$ (if the distributed algorithm requires

[1] There is a small nuance: to simulate the second round of $\mathcal{A}$ on u, a vertex at distance ℓ from v, we might need to simulate the first round of $\mathcal{A}$ on u's neighbors, which may be at distance $\ell + 1$ from v. But for the purpose of the simulation, we can assume that u has no neighbors that are at distance $\ell + 1$ from v, as anything that happens to u after the first round cannot affect v. Therefore, once we have uncovered the ℓ-neighborhood of v, we can ignore the rest of the graph.

ℓ rounds). We show that this analysis is far from tight, in fact these complexity measures are all $O(\log n)$.[2] In other words, Algorithm 1 can be implemented as an $(O(\log n), O(\log n), 0, O(\log n), \frac{1}{n^4})$-LCA. We note that in order to guarantee that the correctness proof still holds, the men need to propose to the women. The more general result is formally stated below as Theorem 3.2.2; we prove it by bounding the size of the ℓ-neighborhood of v, for any $v \in V$.

The results of the following subsection are stated for d-light graphs; they also hold (without modification of the proofs) for (polylog n)-conditionally d-light graphs.

3.2 Bounding the Neighborhood Size

Recall that $N_i(v)$ is the set of vertices at distance at most i from v.

Claim 3.2.1. *For any integer $i > 0$, there exists a constant c_i such that for any d-light graph $G = (V, E)$ and vertex $v \in V$, it holds that* $\Pr[|N_i(v)| \leq c_i \log n] \geq 1 - \frac{1}{n^4}$.

Proof. Let $\mathcal{N}^i$ be the random variable representing the number of vertices in the i-neighborhood of vertex v. We prove by induction that $\Pr[\mathcal{N}^i \leq c_i \log n] \geq 1 - \frac{i}{n^5}$, where $c_i = (4ed)^{i-1}c_1$.

For the base, from Property 2.3.11, taking $S = \{v\}$, it holds that there exists a constant c_1 such that , $\Pr[\mathcal{N}^1 > c_1 \log n] \leq 1/n^5$

Assuming that the claim holds for all integers smaller than i, we show that it holds for i. We use the law of total probability.

$$\begin{aligned}
\Pr[\mathcal{N}^i > c_i \log n] =& \Pr[\mathcal{N}^i > c_i \log n | \mathcal{N}^{i-1} \leq c_{i-1} \log n] \Pr[\mathcal{N}^{i-1} \leq c_{i-1} \log n] \\
&+ \Pr[\mathcal{N}^i > c_i \log n | \mathcal{N}^{i-1} > c_{i-1} \log n] \Pr[\mathcal{N}^{i-1} > c_{i-1} \log n] \\
\leq& \Pr[\mathcal{N}^i > c_i \log n | \mathcal{N}^{i-1} \leq c_{i-1} \log n] + \Pr[\mathcal{N}^{i-1} > c_{i-1} \log n] \\
\leq& \frac{1}{n^5} + \frac{i-1}{n^5}.
\end{aligned}$$

where the last inequality uses Property 2.3.11 and the inductive hypothesis. □

We conclude the following.

Theorem 3.2.2. *Let $G = (V, E)$ be a d-light graph, where $d > 0$ is a constant, let O be some finite set and let $F : V \to O$ be some function on the vertices. As-*

[2] ℓ appears exponentially in the constant.

sume that $\mathcal{A}$ is a crisp constant-time distributed algorithm for F. Then there is an $(O(\log n), O(\log n), 0, O(\log n), \frac{1}{n^4})$-LCA for F.

Proof. From Claim 3.2.1, the ℓ-neighborhood of any vertex is number of probes required to simulate the distributed algorithm is $O(\log n)$, for some constant $c > 0$, with probability at least $\frac{1}{n^5}$. Assume that the distributed algorithm $\mathcal{A}$ runs for ℓ rounds. As $\mathcal{A}$ is crisp, simulating it on this neighborhood requires time $\ell \cdot O(1) \cdot O(\log n)$, and this is clearly also an upper bound on the required transient memory. A union bound gives the required failure probability. □

Chapter 4

Reducing LCAs to Online Algorithms

In the previous chapter we showed a reduction of LCAs to distributed algorithms. In this chapter, we show a similar reduction to online algorithms. This reduction, however, is much more involved, and requires a more sophisticated set of tools for analysis. The idea is simple - we generate a random order on the vertices, and simulate an online algorithm on this order. Two major problems arise: how do we bound the number of probes the algorithm makes to the graph, and how do we store this randomness in polylogarithmic space?

This chapter is based mainly on [90], but also in part on [4] and [67].

4.1 Preliminaries

For simplicity, we only consider online graph algorithms on vertices; an analogous definition holds for algorithms on edges, and our results hold for them as well. Intuitively, such an algorithm is presented with the vertices of a graph $G = (V, E)$ in some arbitrary order. Once the algorithm is presented with a vertex v (as well as the edges to the neighbors of v that arrived before v), it must irrevocably output a value which we denote $f(v)$. The output of the algorithm is the combination of all of these intermediate outputs, namely the function $f : V \to O$ (where O is some arbitrary finite set). The correctness of our LCAs is immediate (from the correctness of the global (online) algorithm); the focus of this chapter is therefore proving the probe, memory (enduring and transient) and time complexity.

Algorithm 2: Online (Greedy) MIS Algorithm

Input : $G = (V, E)$ and vertex permutation π
Output: An independent set S

$I \leftarrow \emptyset$;
Assume the vertices are ordered $v_1, v_2, \ldots, v_n$ in π;
for $i \leftarrow 1$ *to* n **do**
  if $\forall u \in N(v_i)$, $u \notin I$ **then**
    $I \leftarrow I \cup \{v_i\}$;

We require that the online algorithm will be neighborhood dependent in the sense that the value $f(v)$ is only a function of the values $\{f(u)\}$ for the neighbors u of v that the algorithm has already seen. Formally,

Definition 4.1.1 (Neighborhood-dependent online graph algorithm)**.** *Let $G = (V, E)$ be a graph, and let O be some arbitrary finite set. A* neighborhood-dependent online graph algorithm *$\mathcal{A}$ takes as input a vertex $v \in V$ and a sequence of pairs $\{(u_1, o_1), \ldots (u_\ell, o_\ell)\}$ where $\forall i, u_i \in V, o_i \in O$, and outputs a value $o \in O$. For every permutation Π of the vertices in V, define the output of $\mathcal{A}$ on G with respect to Π as follows. Denote by v_i^Π the vertex at location i under Π. Define $f(v_i^\Pi)$ recursively by invoking $\mathcal{A}$ on v_i^Π and the sequence of values $(v_j^\Pi, f(v_j^\Pi))$, such that $j < i$ and $(v_j^\Pi, v_i^\Pi) \in E$.*[1]

Let R be a search problem on graphs. We say that $\mathcal{A}$ is a neighborhood-dependent online graph algorithm for R if for every graph G and every permutation Π, the output of $\mathcal{A}$ on G with respect to Π satisfies the relation defining R.

For example, consider the following online algorithm for computing an MIS, Algorithm 2. The algorithm is essentially the following: Initialize the set $I = \emptyset$. When a vertex v arrives, the algorithm checks whether any of v's neighbors is in I. If none of them is, v is added to I. Otherwise, v is not in I. Algorithm 2 is clearly neighborhood-dependent; furthermore, it is crisp. The reader might find it useful to refer to this algorithm while reading this chapter, as it is simple yet illustrative. It is easy to see that many other online algorithms on graphs, such as the greedy algorithms for vertex coloring and maximal matching, and many load balancing algorithms (see Chapter 9), are also neighborhood-dependent.

We wish obtain an LCA $\mathcal{A}$ for MIS. The high-level idea of the reduction is the

[1] Note that in a general online algorithm, we only require that $j < i$ and $(v_j^\Pi, f(v_j^\Pi))$.

following. $\mathcal{A}$ receives as input a graph $G = (V, E)$. The first time it is invoked, it samples a hash function $h : V \to Q$, where Q is some finite set, and writes h to the enduring memory. The function h is used to define some ordering π on the vertices (we will elaborate extensively on how this is done). When $\mathcal{A}$ is queried on a vertex v, it simulates Algorithm 2 on G, where the vertices arrive in the order of π. It replies "yes" if and only if Algorithm 2 determines that v is in the MIS.

4.1.1 Static and Adaptive k-wise Independence

Our probe and time bounds rely on the fact that we can generate a random order on the vertices. Storing a random order over all of the vertices would require $\Omega(n \log n)$ enduring memory; in order to only use polylogarithmic memory, we use a hash function that guarantees that our ordering is "sufficiently random".

Definition 4.1.2 (k-wise independent hash functions)**.** *For $n, L, k \in \mathbb{N}$ such that $k \leq n$, a family of functions $\mathcal{H} = \{h : [n] \to [L]\}$ is* k-wise independent *if for all distinct $x_1, x_2, \ldots, x_k \in [n]$, when H is sampled uniformly from $\mathcal{H}$ we have that the random variables $H(x_1), H(x_2), \ldots, H(x_k)$ are independent and uniformly distributed in $[L]$.*

To quantify what we mean by "almost" k-wise independence, we use the notion of *statistical distance*.

Definition 4.1.3 (Statistical distance)**.** *For random variables X and Y taking values in $\mathcal{U}$, their* statistical distance *is*

$$\Delta(X, Y) = max_{D \subseteq \mathcal{U}} |\Pr[X \in D] - \Pr[Y \in D]|.$$

For $\epsilon \geq 0$, we say that X and Y are ϵ-close if $\Delta(X, Y) \leq \epsilon$.

Definition 4.1.4 (ϵ-almost k-wise independent hash functions)**.** *For $n, L, k \in \mathbb{N}$ such that $k \leq n$, let Y be a random variable sampled uniformly at random from $[L]^k$. For $\epsilon \geq 0$, a family of functions $\mathcal{H} = \{h : [n] \to [L]\}$ is* ϵ-almost k-wise independent *if for all distinct $x_1, x_2, \ldots, x_k \in [n]$, we have that $\langle H(x_1), H(x_2), \ldots, H(x_k)\rangle$ and Y are ϵ-close, when H is sampled uniformly from $\mathcal{H}$. We sometimes use the term "ϵ-dependent" instead of "ϵ-almost independent".*

Another interpretation of ϵ-almost k-wise independent hash functions is as functions that are indistinguishable from uniform for a static distinguisher that is allowed to query

the function in at most k places. In other words, we can imagine the following game: the (computationally unbounded) distinguisher $\mathcal{D}$ selects k inputs $x_1, x_2, \ldots, x_k \in [n]$, and gets in return values $F(x_1), F(x_2), \ldots, F(x_k)$, where $F : [n] \to [L]$ is either chosen from $\mathcal{H}$ (which we denote by $\mathcal{D}^{F \leftarrow \mathcal{H}}$), or is selected uniformly at random. $\mathcal{H}$ is ϵ-almost k-wise independent if no such $\mathcal{D}$ can differentiate the two cases with advantage larger than ϵ. In this paper we will need to consider *adaptive* distinguishers that can select each x_i based on the values $F(x_1), F(x_2), \ldots, F(x_{i-1})$.

Definition 4.1.5 (ϵ-almost adaptive k-wise independent hash functions)**.** *For $n, L, k \in \mathbb{N}$ such that $k \leq n$ and for $\epsilon \geq 0$, a family of functions $\mathcal{H} = \{h : [n] \to [L]\}$ is* ϵ-almost adaptive k-wise independent *if for every (computationally unbounded) distinguisher $\mathcal{D}$ that makes at most k queries to an oracle F it holds that*

$$|\Pr[\mathcal{D}^{F \leftarrow \mathcal{H}} = 1] - \Pr[\mathcal{D}^{F \leftarrow \mathcal{G}} = 1]| \leq \epsilon,$$

where $\mathcal{G}$ is the set of all functions $F : [n] \to [L]$.

We say that $\mathcal{H}$ is adaptive k-wise independent *if it is 0-almost adaptive k-wise independent.*

Maurer and Pietrzak [71] showed a very efficient way to transform a family of (static) almost k-wise independent functions into a family of adaptive almost k-wise independent functions with similar parameters. For our purposes, it is enough to note that every family of (static) almost k-wise independent function is in itself also adaptive almost k-wise independent. While the parameters deteriorate under this reduction, they are still good enough for our purposes. We provide the reduction here for completeness.

Lemma 4.1.6. *[71] For $n, L, k \in \mathbb{N}$ such that $k \leq n$ and for $\epsilon \geq 0$, every family of functions $\mathcal{H} = \{h : [n] \to [L]\}$ that is ϵ-almost k-wise independent is also adaptive ϵL^k-almost k-wise independent.*

Proof. The proof is by a simulation argument. Consider an adaptive distinguisher $\mathcal{D}$ that makes at most k queries; assume without loss of generality that $\mathcal{D}$ always makes *exactly* k distinct queries. We can define the following static distinguisher $\mathcal{D}'$ with oracle access to some function F as follows: $\mathcal{D}'$ samples k distinct outputs $y_1, y_2, \ldots, y_k \in [L]$ uniformly at random. $\mathcal{D}'$ then simulates $\mathcal{D}$ by answering the ith query x_i of $\mathcal{D}$ with y_i. Let σ be the bit $\mathcal{D}$ would have output in this simulation. Now $\mathcal{D}'$ queries F

for $x_1, x_2, \ldots x_k$ (note that $\mathcal{D}'$ makes all of its queries simultaneously and is therefore static). If the replies obtained are consistent with the simulation (i.e. $y_i = F(x_i)$ for every i) then $\mathcal{D}'$ outputs σ. Otherwise, it outputs 0. By definition,

$$\Pr[{\mathcal{D}'}^{F \leftarrow \mathcal{H}} = 1] = L^{-k} \Pr[\mathcal{D}^{F \leftarrow \mathcal{H}} = 1].$$

This implies that for every H and G

$$|\Pr[{\mathcal{D}'}^{F \leftarrow \mathcal{H}} = 1] - \Pr[{\mathcal{D}'}^{F \leftarrow \mathcal{G}} = 1]| = L^{-k} |\Pr[\mathcal{D}^{F \leftarrow \mathcal{H}} = 1] - \Pr[\mathcal{D}^{F \leftarrow \mathcal{G}} = 1]|.$$

The proof follows. □

In particular, Lemma 4.1.6 implies that k-wise independent functions are also adaptive k-wise independent, and we get the following theorem:

Theorem 4.1.7 (cf. [106] Proposition 3.33 and Lemma 4.1.6)**.** *For $n, k \in \mathbb{N}$ such that $k \leq n$ and n is a power of 2, there exists a family of functions $\mathcal{H} = \{h : [n] \to [n]\}$ that is adaptive k-wise independent, whose seed length is $k \log n$. The time required to evaluate each h is $O(k)$ word operations and the memory required per evaluation is $O(\log n)$ bits (in addition to the memory for the seed).*

Naor and Naor [75] showed that relaxing from k-wise to almost k-wise independence can imply significant savings in the family size. As we we also care about the evaluation time of the functions, we will employ a recent result of Meka et al., [72] (which, using Lemma 4.1.6, also applies to *adaptive* almost k-wise independence). The following is specialized from their work to the parameters we mostly care about in this work.

Theorem 4.1.8 ([72] and Lemma 4.1.6)**.** *For every $n, L, k \in \mathbb{N}$ and $\epsilon > 0$ such that n and L are powers of 2, and such that $k \cdot L = O(\log n)$ and $1/\epsilon = poly(n)$, there is a family of adaptive ϵ-almost k-wise independent functions $\mathcal{H} = \{h : [n] \to [L]\}$ such that choosing a random function from $\mathcal{H}$ takes $O(\log n)$ random bits. The time required to evaluate each h is $O(\log k)$ word operations and the (enduring) memory is $O(\log n)$ bits.*

The property of almost k-wise independent hash functions H that we will need is that an algorithm querying H will not query "too many" preimages of any particular output of H. Specifically, if the algorithm queries H more than $cL \log n$ times, none of the values will appear more than twice their expected number. More formally:

Proposition 4.1.9. *There exists a constant c such that following holds. Let $n, L, k \in \mathbb{N}$ be such that $k \leq n$ and let $\epsilon \geq 0$. Let $\mathcal{H} = \{h : [n] \to [L]\}$ be a family of functions that is ϵ-almost adaptive k-wise independent. Let $\mathcal{A}$ be any procedure with oracle access to H sampled uniformly from $\mathcal{H}$ and let ℓ be any value in $[L]$. Define the random variable m to be the number of queries $\mathcal{A}$ makes and the random variables $x_1, x_2, \ldots, x_m$ to be those queries. Then conditioned on $cL \log n \leq m \leq k$, it holds that:*

$$\Pr[|\{x_i | H(x_i) = \ell\}| > 2m/L] \leq \frac{1}{2n^5} + \epsilon.$$

Proof. Consider any $\mathcal{A}$ as in the theorem and assume for the sake of contradiction that

$$\Pr[|\{x_i | H(x_i) = \ell\}| > 2m/L \mid cL \log n \leq m \leq k] > \frac{1}{2n^5} + \epsilon.$$

Consider $\mathcal{A}$ with access to a uniformly selected $G : [n] \to [L]$. Define m' to be the number of queries $\mathcal{A}$ makes in such a case and let $x'_1, \ldots, x'_{m'}$ be the set of queries. By the Chernoff bound, conditioned on any fixing of m', we have that $\Pr[|\{x'_i | H(x'_i) = \ell\}| > 2m'/L]$ is exponentially small in m'/L. Therefore, by a union bound, for a sufficiently large c we have that $\Pr[|\{x_i | H(x_i) = \ell\}| > 2m'/L \mid cL \log n \leq m' \leq k] \leq \frac{1}{2n^5}$ (note that $L < n$, otherwise the theorem is trivially true). We thus have that $\mathcal{A}$ distinguishes the distribution H from the distribution G with k queries, with advantage ϵ, in contradiction to H being ϵ-almost adaptive k-wise independent. □

We study the application of adaptive ϵ-almost k-wise independent functions to d-light graphs extensively in this chapter. We do not explicitly give the dependence of ϵ on n, d and k, although we assume that $\epsilon = \frac{1}{\text{poly}\, n}$ and (in Subsection 4.4), that $\epsilon < \frac{1}{d^{2k}}$.

4.2 Almost k-wise Random Orderings

Our LCAs will emulate the execution of online algorithms (that are neighborhood dependent as in Definition 4.1.1). One obstacle is that the output of an online algorithm may depend on the order in which it sees the vertices of its input graph (or the edges, in case we are considering an algorithm on edges). As the combined output of an LCA on all vertices has to be consistent, it is important that in all of these executions the algorithm uses the same permutation. Choosing a random permutation on n vertices requires $\Omega(n \log n)$ bits which is disallowed (as it will imply enduring memory

$\Omega(n \log n)$). Instead, we would like to have a derandomized choice of a permutation that will be "good enough" for the sake of our LCAs and can be sampled using as few as $O(\log n)$ bits.

Past works on LCAs (e.g., [4, 42, 67, 68]) have used k-wise and almost k-wise independent orderings to handle the derandomized ordering of vertices. A family of ordering $\mathbf{\Pi} = \{\Pi_r\}_{r \in R}$, indexed by R is *k-wise independent* if for every subset $S \subseteq [n]$ of size k, the projection of $\mathbf{\Pi}$ onto S (denoted by $\mathbf{\Pi}(S)$) is uniformly distributed over all $k!$ possible permutations of S. Denote this uniform distribution by $U(S)$. We have that $\mathbf{\Pi}$ is *ϵ-almost k-wise independent* if for every k-element subset S we have that $\Delta(\mathbf{\Pi}(S), U(S)) \leq \epsilon$ (where Δ is the statistical distance, see Definition 4.1.3). One can give adaptive versions of these definitions (in the spirit of Definition 4.1.5). The following "warm up" theorem shows that if $L = \text{poly}(n)$ and $\mathcal{H} = \{h : [n] \to [L]\}$ is k-wise independent, then $\mathbf{\Pi} = \{\Pi_h\}_{h \in \mathcal{H}}$ is a family of $1/\text{poly}(n)$-almost k-wise independent orderings, and that this requires a seed of length $O(k \log n)$. It is easy to show that k-wise independent functions (or even almost k-wise independent functions), directly give ϵ-almost k-wise independent orderings.

Lemma 4.2.1. *For every $n, L, k \in \mathbb{N}, \epsilon > 0$, such that $k \leq n$ and $L \geq k^2/\epsilon$, if $\mathcal{H} = \{h : [n] \to [L]\}$ is k-wise independent then $\mathbf{\Pi} = \{\Pi_h\}_{h \in \mathcal{H}}$ is a family of ϵ-almost k-wise independent ordering.*

Proof. Fix the set S. There is probability smaller than ϵ on the choice of $h \in \mathcal{H}$ that there exist two distinct values i and j in S such that $h(i) = h(j)$. Conditioned on such collision not occurring the order is uniform by the definition of k-wise independent hashing. □

Lemma 4.2.1 and Theorem 4.1.7 imply Theorem 4.2.2.

Theorem 4.2.2. *For every $n, k \in \mathbb{N}, \epsilon > 0$, such that $k \leq n$, there exists a construction of ϵ-almost k-wise independent random ordering of $[n]$ whose seed length is $O(k \log n)$.*

This could potentially be further improved, but one can observe that a lower bound on the seed length of almost $\log n$-wise independent ordering is $\Omega(\log n \log \log n)$. We now show how to define derandomized orderings that require seed $O(\log n)$ and, while not being almost $\log n$-wise independent, are still sufficiently good for our application.

4.3 Upper Bounding the Number of Probes

Let us now consider what happens if we let L be much smaller, namely a constant. This will reduce the seed length to $O(k + \log n)$ (when we let $\mathcal{H}$ be *almost* k-wise independent). However, we will lose the k-wise almost independence (for sub-constant error) of the ordering: Consider two variables $i < j$. Conditioned on $h(i) \neq h(j)$, the order of i and j under Π_h is uniform. But with constant probability $h(i) = h(j)$ and then, according to Definition 4.3.2, i will come before j under Π_h (recall that the ordering is based on the labels $(h(i), i)$). Nevertheless, even though $\mathbf{\Pi}$ is no longer $1/n$-almost k-wise independent, we will show that a constant L suffices for our purposes. To formally define what this means we introduce some definitions.

Definition 4.3.1 (Level, Levelhood)**.** *Let $G = (V, E)$ be a graph. Let $h : V \to \mathbb{N}$ be a function that assigns each vertex an integer. For each $v \in V$, we call $h(v)$ the* level *of v. Denote the restriction of a set of vertices $S \subseteq V$ to only vertices of a certain level ℓ, $\{x \in S : h(x) = \ell\}$, by $S\|_\ell$.*

Let $S \subseteq V$, and let $N_\ell(S)$ be the neighbors of S that are in level ℓ. That is $N_\ell(S) = \{u \in V : u \in N(S), h(u) = \ell\}$. The ℓ^{th} levelhood *of S (denoted $\Psi_\ell(S)$), is defined recursively as follows. $\Psi_\ell(\emptyset) = \emptyset$. $\Psi_\ell(S) = \{S \cup \Psi_\ell(N_\ell(S))\}$. In other words, we initialize $\Psi_\ell(S) = S$, and add to $\Psi_\ell(S)$ neighbors of level ℓ, until we cannot add any more vertices.*

We define the *rank* of a vertex to be the concatenation of its level and ID. Formally,

Definition 4.3.2 (Ranking)**.** *Let L be some positive integer. A function $r : [n] \to [L]$ is a* ranking *of $[n]$, where $r(i)$ is called the* level *of i. The* ordering *Π_r which corresponds to r is a permutation on $[n]$ obtained by defining $\Pi_r(i)$ for every $i \in [n]$ to be its position according to the monotone increasing order of the relabeling $i \mapsto (r(i), i)$. In other words, for every $i, j \in [n]$ we have that $\Pi_r(i) < \Pi_r(j)$ if and only if $r(i) < r(j)$ or $r(i) = r(j)$ and $i < j$. The pair $(r(i), i)$ is called the* rank *of i.*

Informally, vertices of higher rank "arrive earlier".

Definition 4.3.3 (Relevant vicinity)**.** *The* relevant vicinity *of a vertex v, denoted $\Im_h(v)$ (relative to a hash function $h : V \to [L]$), is defined constructively as follows. Let Π_h be the permutation defined by h, as in Definition 4.3.2. Initialize $\Im_h(v) = \{v\}$. For each $u \in \Im_h(v)$, add to $\Im_h(v)$ all vertices $w \in N(u) : \Pi_h(w) < \Pi_h(u)$. Continue*

adding vertices until no more can be added. The relevant vicinity of a set of vertices U is the union of the relevant vicinities of the vertices in U.

The relevant vicinity of a vertex v is exactly the vertices that our LCA will simulate the online algorithms on when queried about v.[2] Because we cannot make any assumptions about the original labeling of the vertices, we upper bound the size of the relevant vicinity by defining the *containing vicinity*, where we assume that the worst case always holds. That is, if two neighbors have the same level, the one that is queried first appears before the other in the permutation. The size of the containing vicinity is clearly an upper bound on the size of a relevant vicinity, for the same hash function.

Definition 4.3.4 (Containing vicinity)**.** *The* containing vicinity *of a vertex v (relative to a hash function $h : V \to [L]$), given that $h(v) = \ell$, is $\Psi_L(\Psi_{L-1}(\ldots\Psi_{\ell+1}(\Psi_\ell(v))\ldots))$. In other words, let S_ℓ be the ℓ^{th} levelhood of v, and for $i \in \{\ell+1, \ldots, L\}$, $S_i = \Psi_i(S_{i-1})$. The containing vicinity is then S_L. The containing vicinity of a set of vertices U is the union of the containing vicinities of the vertices in U.*

4.3.1 Upper Bounding the Size of the Relevant Vicinity

Lemma 4.3.5. *Let $G = (V, E)$ be a d-light graph, $|V| = n$, and let L be a (constant) integer such that $L > 24d$. Let c be such that Proposition 4.1.9 and Property 2.3.10 hold, and let $\kappa = 2^L c \log n$. Let $k = 6d\kappa$, and let h be an adaptive ϵ-almost k-wise independent hash function, $h : V \to [L]$. For any adaptively exposed set of vertices $U \subseteq V$, whose size is at most $c \log n$, the relevant vicinity of U has size at most κ with probability at least $1 - \frac{1}{n^4} + \frac{1}{n^5}$.*

Corollary 4.3.6. *Let the conditions of Lemma 4.3.5 hold. For any vertex $v \in G$, the relevant vicinity of v has size at most κ with probability at least $1 - \frac{1}{n^4} + \frac{1}{n^5}$.*

The following claim will help prove Lemma 4.3.5.

Claim 4.3.7. *Let the conditions of Lemma 4.3.5 hold; assume without loss of generality that for all $v \in U$, $h(v) = 1$, and let $S_0 = U, S_1 = \Psi_1(S_0), \ldots, S_{\ell+1} =$*

[2] In specific cases, better implementations exist that do not need to explore the entire relevant vicinity. For example, once an LCA for maximal independent set sees that a neighbor of the inquired vertex is in the independent set, it knows that the inquired vertex is not (and can halt the exploration). Nevertheless, in order not to make any assumptions on the online algorithm, we assume that the algorithm explores the entire relevant vicinity.

$\Psi_{\ell+1}(S_\ell), \ldots, S_L = \Psi_L(S_{L-1})$. *Then for all* $0 \le i \le L$,

$$\Pr[|S_i| \le 2^i c \log n \wedge |S_{i+1}| \ge 2^{i+1} c \log n] \le \frac{2}{n^5}.$$

Proof. Fix i and denote the bad event for i as B_i. That is, $B_i \equiv |S_i| \le 2^i c \log n \wedge |S_{i+1}| \ge 2^{i+1} c \log n$. We make the following observation:

$$B_i \Rightarrow |S_{i+1}| > 2^{i+1} c \log n \wedge |S_{i+1}\|_{i+1}| > \frac{|S_{i+1}|}{2}, \tag{4.1}$$

because $S_{i+1}\|_{i+1} = S_{iF+1} \setminus S_i$. In other words, this means that if B_i occurs then the majority of elements in S_{i+1} must have come from the $(i+1)^{th}$-levelhood of S_i.

For all $0 < i \le L$ let $T_i = S_i \cup N(S_i)$. Define two bad events:

$$B_i^1 \equiv |S_{i+1}| > 2^{i+1} c \log n \wedge |T_{i+1}| > 6d|S_{i+1}|$$
$$B_i^2 \equiv |S_{i+1}| > 2^{i+1} c \log n \wedge |T_{i+1}\|_{i+1}| > \frac{12d}{L}|S_{i+1}|$$

From Equation (4.1), the definition of L, and the fact that $T_{i+1}\|_{i+1} = S_{i+1}\|_{i+1}$, we get $B_i \Rightarrow B_i^2$.

By Property 2.3.10,

$$\Pr[B_i^1] \le \frac{1}{n^5}, \tag{4.2}$$

because S_{i+1} is an adaptively exposed subset, $T_{i+1} = S_{i+1} \cup N(S_{i+1})$ and the size of S_{i+1} satisfies the conditions of the proposition. Given $|S_{i+1}| > 2^{i+1} c \log n$, it holds that $|T_{i+1}| > 2^{i+1} c \log n$ because $S_{i+1} \subseteq T_{i+1}$. Also note that T_{i+1} was defined based on the value of h on elements in T_{i+1} only. Therefore, the conditions of Proposition 4.1.9 hold for $|T_{i+1}|$.

$$\begin{aligned}
\Pr[B_i^2 : \neg B_i^1] &\le \Pr[|T_{i+1}\|_{i+1}| > \frac{12d}{L}|S_{i+1}| : |T_{i+1}| \le 6d|S_{i+1}|] \\
&\le \Pr[|T_{i+1}\|_{i+1}| > \frac{2|T_{i+1}|}{L}] \\
&\le \frac{1}{n^5},
\end{aligned} \tag{4.3}$$

where the last inequality is due to Proposition 4.1.9,

Because

$$\Pr[B_i^2] = \Pr[B_i^2 : B_i^1]\Pr[B_i^1] + \Pr[B_i^2 : \neg B_i^1]\Pr[\neg B_i^1],$$

from Equations (4.2) and (4.3), the claim follows. □

Proof of Lemma 4.3.5. To prove Lemma 4.3.5, we need to show that

$$\Pr[|S_L| > 2^L c \log n] < \frac{1}{n^4} - \frac{1}{n^5},$$

We show that for $0 \leq i \leq L, \Pr[|S_i| > 2^i c \log n] < \frac{2i}{n^5}$, by induction. For the base of the induction, $|S_0| = 1$, and the claim holds. For the inductive step, assume that $\Pr[|S_i| > 2^i c \log n] < \frac{2i}{n^5}$. Then

$$\begin{aligned}\Pr[|S_{i+1}| > 2^{i+1} c \log n] = \Pr[|S_{i+1}| > 2^{i+1} c \log n : |S_i| > 2^i c \log n]\Pr[|S_i| > 2^i c \log n] \\ + \Pr[|S_{i+1}| > 2^{i+1} c \log n : |S_i| \leq 2^i c \log n]\Pr[|S_i| \leq 2^i c \log n].\end{aligned}$$

From the inductive step and Claim 4.3.7, using the union bound, the lemma follows. □

From Lemma 4.3.5 and Property 2.3.10, we immediately get

Corollary 4.3.8. *Let $G = (V, E)$ be a d-light graph, where $|V| = n$, and let L be an integer such that $L > 24d$. Let c be such that Proposition 4.1.9 and Property 2.3.10 hold, and let $\kappa = 2^L c \log n$. Let $k = 6d\kappa$, and let h be an adaptive k-wise ϵ-almost independent hash function. For any vertex $v \in G$, let S_L be the relevant vicinity of v. Then*

$$\Pr[|\{(u, w) \in E \mid u \in S_L\}| > k] < 1/n^4.$$

Corollary 4.3.8 essentially shows the following: Assume that $G = (V, E)$ is a d-light graph, and there is some function $F : V \to \mathcal{R}$ that is computable by a neighborhood-dependent online algorithm $\mathcal{A}$. Then, in order to compute $F(v)$ for any vertex $v \in V$, we only need to look at a logarithmic number of vertices and edges with high probability. Note that in order to calculate the relevant vicinity, we need to look at all the vertices in the relevant vicinity and all of their neighbors (to make sure that we have not overlooked any vertex). This is upper bound by the number of edges which have an endpoint in the relevant vicinity, as the relevant vicinity is connected. Furthermore, as we will see in Section 4.5, we would like to store the subgraph induced by the relevant vicinity, and for this, we need to store all of the edges.

Applying a union bound over all the vertices gives that the number of probes we need to make per query (i.e., each time the LCA is queried about a vertex) is $O(\log n)$ with probability at least $1 - 1/n^3$ even if we are queried about all the vertices.

4.4 Expected Size of the Relevant Vicinity

In Section 4.5, we show constructions of the subgraph induced by the relevant vicinity whose running times and transient memory requirements are dependent on t_v and t_e, the size of the relevant vicinity and the number of edges adjacent to the relevant vicinity, respectively. All the dependencies can be upper bound by $O(t_e^2)$. In this section, we prove that the expected value of t_e^2 in a d-light graph is a constant (when d is a constant).

Proposition 4.4.1. *For any d-light graph $G = (V, E)$ and any vertex $v \in V$, the expected number of simple paths of length t originating from v is at most d^t.*

We prove a slightly more general claim, from which Proposition 4.4.1 immediately follows (taking S in the proposition to be the empty set).

Claim 4.4.2. *For any d-light graph $G = (V, E)$, any adaptively exposed subset $S \subseteq V$, and any vertex $v \in N(S) \setminus S$ (or any vertex v if S is empty), the expected number of simple paths of length t originating from v and not intersecting with S is at most d^t.*

Proof. The proof is by induction on t. For the base of the induction, $t = 0$, and there is a single simple path (the empty path). For the inductive step, let $t > 0$ and assume that the claim holds for $t - 1$. We show that it holds for t. Given S, let $S' = S \cup \{v\}$. Let a be a random variable representing the number of neighbors of v that are not in S; that is, $a = |N(v) \setminus S|$. Because G is d-light, $\mathbb{E}(a) \leq d$. Fixing a, label the neighbors of v that are not in S by $w_1, w_2, \ldots, w_a$. By the inductive hypothesis (as S' is also adaptively exposed), for all $i = 1, \ldots, a$, the expected number of simple paths of length $t - 1$ originating from w_i and not intersecting with S' is at most d^{t-1}. The expected number of simple paths of length t originating from v and not intersecting with S is therefore upper bounded by

$$\sum_j \Pr[a = j] j \cdot d^{t-1} = \mathbb{E}[a] d^{t-1} \leq d^t.$$

□

For any simple path p originating at some vertex v, we would like to determine whether all the vertices on p are in the relevant vicinity of v. As we cannot make any assumptions about the original labels of the vertices, we upper bound this by the probability that the levels of the vertices on the path are non-decreasing.

Definition 4.4.3 (Legal path)**.** *We say that a path $p = v \rightsquigarrow u$ is* legal *if it is simple and the labels of the vertices on p are in non decreasing order.*

Definition 4.4.4 (Prefix-legal path)**.** *We say that a path $p = v \rightsquigarrow u$ of length t is* prefix-legal *if it is simple, and the prefix of p of length $t-1$ is legal.*

Proposition 4.4.5. *Let $G = (V, E)$ be a d-light graph and let the conditions of Lemma 4.3.5 hold. That is, $k, \kappa = O(\log n)$ and $L > 24d$. For any $c > 0$ there exists a value $L = O(d)$ for which the following holds: Let h be an adaptive k-wise ϵ-almost independent hash function, $h : V \to [L]$. For any path p of length $t' < k - 1$ originating at some vertex v, the probability that p is legal is at most $d^{-ct'} + \epsilon$.*

Proof. For any simple path p of length t' from v, there are $|L|^{t'}$ possible values for the levels of the vertices of p. Let the values be $r_0, r_1, \ldots, r_{t'-1}$. We define $t'+1$ new variables $a_0 = r_0$, $a_1 = r_1 - r_0$, $\ldots$, $a_{t'-1} = r_{t'-1} - r_{t'-2}$, $a_{t'} = L - r_{t'-1}$. Clearly, the a_i's uniquely define the r_i's and p is legal if and only if $a_0, \ldots, a_{t'}$ are all non-negative.

Note that the $\sum_{i=0}^{t'} a_i = L$; hence computing the number of possible legal values of $a_0, \ldots, a_{t'}$ is the same as computing the number of ways of placing L identical balls in $t'+1$ distinct bins,[3] where each bin represents a vertex and if there are k balls in bin y, then $r_y = r_{y-1} + k$. This is known to be $\binom{t'+L}{t'}$. Therefore, assuming the choices of the levels are all uniform,

$$\Pr[p \text{ is legal}] = \frac{1}{L^T}\binom{t'+L}{t'} \leq \left(\frac{e(t'+L)}{Lt'}\right)^{t'}.$$

From the definition of ϵ-almost adaptive k-wise independent hash functions, we immediately get

$$\Pr[p \text{ is legal}] \leq \left(\frac{e(t'+L)}{Lt'}\right)^{t'} + \epsilon.$$

The result follows, by selecting an appropriate value for L (dependent on c). □

The following corollary is immediate, setting $t' = t - 1$ in Proposition 4.4.5.

[3] Alternatively, one can view this as placing $t'+1$ separators between L balls.

Corollary 4.4.6. *Let $G = (V, E)$ be a d-light graph and let the conditions of Lemma 4.3.5 hold. That is, $k, \kappa = O(\log n)$ and $L > 24d$. For any $c > 0$ there exists a value $L = O(d)$ for which the following holds: Let h be an adaptive k-wise almost ϵ-independent hash function, $h : V \to [L]$. For any path p of length $t < k$ originating at some vertex v, the probability that p is prefix-legal is at most $d^{c(1-t)} + \epsilon$.*

As a warm-up, we first show that the expected number of edges adjacent to a relevant vicinity is a constant.

Lemma 4.4.7. *Let $G = (V, E)$ be a d-light graph and let the conditions of Proposition 4.4.5 hold. That is, $k, \kappa = O(\log n)$ and $L = O(d)$. Let h be an adaptive k-wise ϵ-almost independent hash function, $h : V \to [L]$. Then the expected number of edges adjacent to the relevant vicinity of a vertex in G is $O(1)$.*

Proof. Let v be any vertex, let S_L be the relevant vicinity of v, and let E_L be the set of edges with at least one endpoint in S_L. Let $(u, w) \in E$ be any edge, and denote by $p^{\leq k}_{(v,u,w)}$ the indicator random variable whose value is 1 if there exists a prefix-legal path of length at most k from v to w whose last edge is (u, w), and 0 otherwise. Similarly, denote by $p^{>k}_{(v,u,w)}$ the random variable whose value is 1 if there exists a prefix-legal path of length greater than k from v to w whose last edge is (u, w), and 0 otherwise. For any $(u, w) \in E$,

$$\Pr[(u, w) \in E_L] \leq \Pr[p^{\leq k}_{(v,u,w)}] + \Pr[p^{>k}_{(v,u,w)}].$$

Therefore,

$$\begin{aligned}
\mathbb{E}[|E_L|] &\leq \sum_{(u,w)\in E} \Pr[(u,w) \in E_L] \\
&\leq \sum_{(u,w)\in E} \Pr[p^{\leq k}_{(v,u,w)}] + \sum_{(u,w)\in E} \Pr[p^{>k}_{(v,u,w)}] \\
&\leq \sum_{(u,w)\in E} \Pr[p^{\leq k}_{(v,u,w)}] + n^2 \Pr[|S_L| > k-1] \\
&\leq \sum_{(u,w)\in E} \Pr[p^{\leq k}_{(v,u,w)}] + 1 \qquad (4.4) \\
&\leq \sum_{\substack{\alpha\text{: paths of} \\ \text{length } \leq k \text{ from } v}} \Pr[\text{path } \alpha \text{ is prefix-legal }] + 1 \\
&\leq \sum_{t\leq k} \left(\sum_{\substack{\alpha\text{: paths of} \\ \text{length } t \text{ from } v}} \Pr[\text{path } \alpha \text{ is prefix-legal }] \right) + 1 \\
&\leq \sum_{t\leq k} (d^{t+c-ct} + \epsilon) + 1 = O(1) \qquad (4.5)
\end{aligned}$$

where Inequality (4.4) is due to Lemma 4.3.5, and Inequality (4.5) is due to Proposition 4.4.1 and Corollary 4.4.6. □

Lemma 4.4.8. *Let $G = (V, E)$ be a d-light graph and let the conditions of Proposition 4.4.5 hold. That is, $k, \kappa = O(\log n)$ and $L = O(d)$. Furthermore let $c > 3$ be a constant. Let h be an adaptive k-wise ϵ-almost independent hash function, $h : V \to [L]$. Denote by E_L the number of edges that have at least one endpoint in the relevant vicinity of some vertex v. Then, $\mathbb{E}[|E_L|^2] = O(1)$.*

Proof. For any edge $e = (u, w) \in E$, let $\mathbb{I}_e$ be an indicator variable whose value is 1 if $e \in E_L$ and 0 otherwise. Let $\mathbb{I}_{|S_L|>k}$ be an indicator variable whose value is 1 if $|S_L| > k$ and 0 otherwise. Then

$$|E_L|^2 = \left(\sum_{e\in E} \mathbb{I}_e \right)^2 = \sum_{e\in E} \mathbb{I}_e \sum_{f\in E} \mathbb{I}_f$$

Let $p^{\leq k}_{(v,u,w)}$ and $p^{>k}_{(v,u,w)}$ be as in the proof of Lemma 4.4.7.

$$\begin{aligned}
|E_L|^2 &\leq \sum_{(u,w)\in E}(p^{\leq k}_{(v,u,w)} + p^{>k}_{(v,u,w)}) \sum_{(x,y)\in E}(p^{\leq k}_{(v,x,y)} + p^{>k}_{(v,x,y)}) \\
&= \sum_{(u,w)\in E}\sum_{(x,y)\in E}(p^{\leq k}_{(v,u,w)} + p^{>k}_{(v,u,w)})(p^{\leq k}_{(v,x,y)} + p^{>k}_{(v,x,y)}) \\
&\leq \sum_{(u,w)\in E}\sum_{(x,y)\in E}(p^{\leq k}_{(v,u,w)}p^{\leq k}_{(v,x,y)} + 3\mathbb{I}_{|S_L|>k-1}) \\
&\leq 3n^4\mathbb{I}_{|S_L|>k-1} + \sum_{(u,w)\in E}\sum_{(x,y)\in E}(p^{\leq k}_{(v,u,w)}p^{\leq k}_{(v,x,y)}). \qquad (4.6)
\end{aligned}$$

For every vertex $u \in V$, let σ^t_u denote the number of simple paths from v to u of length t, and label these paths arbitrarily by $q^t_u(i), i = 1, 2, \ldots \sigma^t_u$. For each path $q^t_u(i)$, let $\hat{q}^t_u(i)$ be the random variable whose value is 1 if $q^t_u(i)$ is prefix-legal, and 0 otherwise. Let Λ^t_v denote the total number of simple paths of length t originating in v.

$$\begin{aligned}
\sum_{(u,w)\in E}\sum_{(x,y)\in E}(p^{\leq k}_{(v,u,w)}p^{\leq k}_{(v,x,y)}) &\leq \sum_{w\in V}\sum_{t\leq k}\sum_{i=1}^{\sigma^t_w}\sum_{y\in V}\sum_{s\leq k}\sum_{j=1}^{\sigma^s_y}\hat{q}^t_w(i)\hat{q}^s_y(j) \\
&\leq 2\sum_{w\in V}\sum_{t\leq k}\sum_{i=1}^{\sigma^t_w}\sum_{y\in V}\sum_{s\leq t}\sum_{j=1}^{\sigma^s_y}\hat{q}^t_w(i)\hat{q}^s_y(j) \qquad (4.7) \\
&= 2\sum_{w\in V}\sum_{t\leq k}\sum_{i=1}^{\sigma^t_w}\hat{q}^t_w(i)\sum_{y\in V}\sum_{s\leq t}\sum_{j=1}^{\sigma^s_y}\hat{q}^s_y(j) \\
&\leq 2\sum_{w\in V}\sum_{t\leq k}\sum_{i=1}^{\sigma^t_w}\hat{q}^t_w(i)\sum_{s\leq t}\Lambda^s_y,
\end{aligned}$$

where Inequality (4.7) is because we order the paths by length and either path can be

longer.

$$\begin{aligned}
\mathbb{E}\left[\sum_{(u,w)\in E}\sum_{(x,y)\in E}(p^{\leq k}_{(v,u,w)}p^{\leq k}_{(v,x,y)})\right] &\leq 2\sum_{t\leq k}\sum_{\substack{\alpha\text{: paths of}\\ \text{length } t \text{ from } v}}\Pr[\alpha \text{ is prefix-legal}]\cdot\mathbb{E}\left[\sum_{s\leq t}\Lambda^s_y\right] \\
&\leq 2\sum_{t\leq k} d^{t+c-ct}\cdot\mathbb{E}\left[\sum_{s\leq t}\Lambda^s_y\right] \\
&\leq 2\sum_{t\leq k}(d^{t+c-ct}+\epsilon)d^{t+1} \\
&\leq 2\sum_{t\leq k}(d^{3t+c-ct}+\epsilon') = O(1). \qquad (4.8)
\end{aligned}$$

From Corollary 4.4.6 we know that we can choose an $L = O(d)$ such that Inequality (4.8) holds. From Inequalities (4.6) and (4.8), Lemma 4.3.5 and the linearity of expectation, the lemma follows. □

4.5 Bounding Running Time and Transient Memory

Let $G = (V, E)$ be a d-light graph, $d > 0$. Let F be a search problem on V, and assume that there exists a neighborhood-dependent online algorithm $\mathcal{A}$ for F. We show how we can use the results of the previous sections to construct an LCA for F. Given an inquiry $v \in V$, we would like to generate a permutation Π on V, build the relevant vicinity of v relative to Π, and simulate $\mathcal{A}$ on these vertices in the order of Π . Because $\mathcal{A}$ is neighborhood-dependent, we do not need to look at any vertices outside the relevant vicinity in order to correctly compute the output of $\mathcal{A}$ on v, and so our LCA will output a reply consistent with the execution of $\mathcal{A}$ on the vertices, if they arrive according to Π. In order to simulate $\mathcal{A}$ on the correct order, we need to store the relevant vicinity and label the vertices in a way that defines the ordering. We show two ways of doing this. The first gives a better time bound, at the expense of a worse space bound. The second gives a better space bound, at the expense of a worse time bound. It remains an open problem whether we can achieve "the best of both worlds"- an LCA requiring $O(\log n \log\log n)$ time and transient space (or even $O(\log n)$). We note that in expectation, both our LCAs require $O(\log\log n)$ time and $O(\log n)$ transient memory.

We define *crispness* as it applies to online algorithms:

Definition 4.5.1. *We say that online algorithm $\mathcal{A}$ is* crisp *if $\mathcal{A}$ requires time and space*

linear in its input length (where the input length is measured by the number of words), and the output of $\mathcal{A}$ is $O(1)$ per query.

Theorem 4.5.2. *Let $G = (V, E)$ be a d-light graph, where $d > 0$ is a constant. Let F be a neighborhood-dependent search problem on V. Assume $\mathcal{A}$ is a crisp neighborhood-dependent online algorithm that correctly computes F on any order of arrival of the vertices. Then*

1. *There is an $(O(\log n), O(\log n \log\log n), O(\log n), O(\log^2 n), 1/n)$-LCA for F.*
2. *There is an $(O(\log n), O(\log^2 n), O(\log n), O(\log n \log\log n), 1/n)$-LCA for F.*

Furthermore, both LCAs require, in expectation, $O(\log\log n)$ time and $O(\log n)$ space.

Proof. We show two methods of constructing an LCA from the online algorithm $\mathcal{A}$. In both, given a query $v \in V$, we use adjacency lists to store the containing vicinity, $\Psi(v)$. We use a single bit to indicate for each vertex, whether it is in the relevant vicinity, $\Im(v)$. This means that for each vertex in $\Im(v)$, we keep a list of all of its neighbors, but for vertices that are in $\Psi(v) \setminus \Im(v)$, we don't need to keep such a list. We denote the number of vertices in the relevant vicinity, $|\Im(v)|$, by t_v, and the total number of edges stored by t_e. Note that $t_e \geq |\Psi(v)| - 1$. This adjacency list representation of $\Psi(v)$ is generated slightly differently in the two constructions. In both cases we label this data structure by $D(v)$. For clarity, we abuse the notation, and use the same name, u, for $u \in V$, and for the vertex which represents u in $D(v)$.

Construction 1. *The first time the LCA is invoked, it chooses a random function h from a family of adaptive k-wise $\frac{1}{2n^5}$-dependent hash functions as in Theorem 4.1.8 and Lemma 4.3.5. The LCA receives as an inquiry a vertex v, and computes $h(v)$. It then discovers the relevant vicinity using DFS. For each vertex u that it encounters, it relabels the vertex $(h(u), u)$. The LCA simulates $\mathcal{A}$ on the vertices arriving in the order induced by the new labels.*

The size of the new label for any vertex u is $|(h(u), u)| = O(\log n)$. In addition, we need to store $\mathcal{A}(u)$ for every vertex $u \in \Im(v)$ (which is $O(1)$ because we assume $\mathcal{A}$ is crisp). Overall, because $|(h(u), u)| = O(\log n)$, the space required for the LCA is upper bounded by $O((t_e + t_v)\log n + t_v)$. From Corollary 4.3.8, $t_v, t_e = O(\log n)$. In expectation, by Lemma 4.4.8, $\mathbb{E}[t_e + t_v] = O(1)$. This gives us the required space bounds. To analyze the running time, we make the following observation.

Observation 4.5.3. *In Construction 1, given $u \in D(v)$, we can access $u \in V$ in $O(1)$.*

We look at each stage of the construction separately:

1. Constructing $D(v)$ is done by DFS, which takes time $O(t_v + t_e)$, as well as the time it takes to generate $|\Im(v)|$ labels, which requires invoking h at most t_e times, and, by Theorem 4.1.8, this requires $O(\text{loglog}\, n)$ time per label.

2. Sorting the labels takes $O(t_v \log t_v)$.

3. Simulating $\mathcal{A}$ on $\Im(v)$ now takes $O(t_e)$ (since $\mathcal{A}$ is crisp).

Given the high probability upper bounds and expected values of t_v and t_e, the first part of the theorem follows. (Note that if $\mathcal{A}$ is not crisp in the sense that computing $F(v)$ is more than linear in the number of neighbors of v, this must be taken into account in the running time.)

In the first construction method, we give each vertex a label of length $O(\log n)$. This seems wasteful, considering we know that the expected size of the relevant vicinity is $O(1)$, and that its size is $O(\log n)$ w.h.p. We therefore give a more space-efficient method of constructing the induced subgraph.

Construction 2. *As in the first method, the first time the LCA is invoked, it chooses a random function h from a family of adaptive k-wise $\frac{1}{2n^5}$-dependent hash functions as in Theorem 4.1.8 and Lemma 4.3.5. Again, we would like to construct the induced subgraph of the relevant vicinity, but to save memory we will not hold the original labels of the vertices (which require $\log n$ bits to represent), but rather new labels that require at most $\text{loglog}\, n$ bits to describe (logarithmic in the size of the relevant vicinity). As before, the LCA receives as an inquiry a vertex v, and computes $h(v)$. We still use $(h(v), v)$ to determine the ordering, however we do not commit this ranking to memory. We initialize $S = \{v\}$, and give v the label 1. In each round i, we look at $N(S)$, and choose the vertex u with the highest rank $(h(u), u)$, out of all the vertices in $N(S)$ which have a lower rank than their neighbor in S. We then add u to S, and give it the label $i + 1$. When we have discovered the entire relevant vicinity, we simulate $\mathcal{A}$ on the vertices in the reverse order of the new labels.*

Note on the required data structure: To efficiently build S, we need to use a slightly different data structure used for storing the adjacency lists than in Construction 1; in fact, we have two adjacency list data structures. The first, $D_1(v)$, contains

only the vertices in the relevant vicinity (not vertices in $\Psi(v) \setminus \Im(v)$). The second, $D_2(v)$, holds the neighbors of the vertices of S which have not yet been added to S. In $D_1(v)$, each vertex is represented by its new label. In $D_2(v)$, each vertex is represented only by its level. We need D_2 to avoid recalculating $h(u)$ more than once for each vertex u. In both $D_1(v)$ and $D_2(v)$, for each edge (i, j) which represents the edge (v_i, v_j), we also store the position of this edge among the edges that leave v_i, and its direction of discovery.

Correctness: Note that v is the vertex of highest rank in its relevant vicinity, and indeed it holds by induction that at step i, the subgraph we expose will contain vertices $v_1 = v, v_2, \ldots v_i$ that have the highest ranking in the relevant vicinity of v (and such that v_i has the ith highest rank). This guarantees the correctness of $\mathcal{A}$ - the reverse order of the labels is exactly the correct ranking of the vertices of the relevant vicinity.

Complexity: The size of the new label for any vertex u is $O(\log t_e)$. In addition, we need to store, for each edge, its position relative to the edges, the edge's direction of discovery, and $\mathcal{A}(u)$ for every vertex $u \in \Im(v)$. Because the graph is d-light, we know that the degree of each vertex is $O(\log n)$ w.h.p., and so keeping the relative position of each edge will require $O(\operatorname{loglog} n)$ bits w.h.p. Overall the space required for $D_1(v)$ is upper bound by $O((t_e + t_v)(\log t_e) + t_v + t_e \operatorname{loglog} n)$. The space required for $D_2(v)$ is $O(t_e \cdot |L|) = O(t_e)$. From Corollary 4.3.8, and Lemma 4.4.8, we have the required space bounds. The expected space bound is due to the length of the seed, $O(\log n)$.

Observation 4.5.4. *In Construction 2, given $u \in D_1(v) \cup D_2(v)$, we can access $u \in V$ in $O(t_v)$.*

Proof. Given $u \in D_1(v)$ (or $D_2(v)$), we find v by DFS from u. As the edges are directed, and $D_1(v)$ and $D_2(v)$ are acyclic, this takes $O(t_v)$. Note that the space required for this DFS may be as much as $t_v \operatorname{loglog} n$, but we use that amount of space regardless. We can store the path $v \rightsquigarrow u$ using the relative locations of the edges, and follow this path on G to find u. □

Similarly to Construction 1, in order to bound the complexity, we look at each stage of the construction separately:

1. Before we add a vertex to S, we need find the vertex u with the lowest $(h(u), u)$ among all vertices in $\Psi(v) \setminus \Im(v)$. This is done by going over all of these vertices to find the minimum, using a DFS on G and the subgraph concurrently, which takes $O(t_e)$.

2. Once we have chosen which vertex u to add to S, we update $D_1(v)$ and $D_2(v)$ to include it. When we look at u's neighbors, though, we don't know whether they are already in $D_1(v)$ or $D_2(v)$, as we don't have pointers to the original vertices in G. From Observation 4.5.4, though, finding this out takes $O(t_v)$ per neighbor, and updating $D_1(v)$ and $D_2(v)$ takes a further $O(1)$ per neighbor.

3. Because of $D_2(v)$, we only need to generate h once for each vertex in $\Psi(v)$. This takes $O(t_e \operatorname{loglog} n)$.

4. Reversing the order of the labels takes $O(t_v)$.

5. Simulating $\mathcal{A}$ on $\Im(v)$ takes $O(t_e)$.

Stages 3, 4 and 5 require $O(t_e \operatorname{loglog} n)$ time in total. Stage 1 accounts for $O((t_e)^2)$, and 2 for $O(t_e t_v)$ overall. Lemma 4.4.8 gives the required expected time bound. Note that we have not discounted the possibility that $\mathcal{A}$ requires the original labels (or "names") of the vertices, in order to compute $F(v)$. When we encounter a vertex, we can always give $\mathcal{A}$ the name of the vertex by exploring the original representation of the graph (the additional time needed is bounded by the time already invested).

The worst case running time and space of the LCA are $O(\log^2 n)$ and $O(\log n \operatorname{loglog} n)$ respectively (w.h.p.). (Note that if $\mathcal{A}$ is not crisp in the sense that $|F(v)|$ is not a constant, this must be taken into account in the space bounds.) □

Our results immediately extend to the case that $d = O(\operatorname{loglog} n)$:

Theorem 4.5.5. *Let $G = (V, E)$ be a d-light graph, where $d = O(\operatorname{loglog} n)$. Let F be a neighborhood-dependent search problem on V. Assume A is a crisp neighborhood-dependent online algorithm that correctly computes F on any order of arrival of the vertices. Then there is an $(O(\operatorname{polylog} n), O(\operatorname{polylog} n), O(\operatorname{polylog} n), O(\operatorname{polylog} n), 1/n)$-LCA for F.*

Next we show that the techniques of the paper thus far do not hold for graphs where the expected degree is $\omega(\operatorname{loglog} n)$, and so the results of this section are, in this sense, tight.

4.6 Tightness with Respect to d-light Graphs

Our results hold for d-light graphs where $d = O(\log\log n)$. We show that at least, using the technique of simulating an online algorithm on a random ordering of the vertices, we cannot do better. We do this by showing that the expected relevant vicinity of the root of a complete binary tree of a d-regular graph is $\Omega(2^{d/2})$, and hence for $d = \omega(\log\log n)$, the expected size will be super-polylogarithmic.

Lemma 4.6.1. *Let T be a complete d-regular binary tree rooted at v, and let Π be a uniformly random permutation on the vertices. The expected size of the relevant vicinity of v relative to Π is at least $2^{d/2}$.*

Proof. Let X_ℓ be a random variable for the number of vertices on level ℓ of the tree that are in the relevant vicinity. There are exactly d^ℓ vertices on level ℓ. The probability that each one is in the relevant vicinity is at least $\frac{1}{\ell!}$, as this is the probability when the permutation on the vertices is truly random.

$$\mathbb{E}[X_\ell] \geq \frac{d^\ell}{\ell!} \geq \left(\frac{d}{\ell}\right)^\ell$$

Taking $\ell = d/2$ gives that $\mathbb{E}[X_\ell] \geq 2^{d/2}$. □

Chapter 5

Approximate Maximum Matching

In this chapter, we use our results from the previous section to give a local computation approximation scheme for maximum matching on graphs of bounded degree. That is, we give an LCA that for any $\epsilon > 0$, computes a maximal matching that is a $(1-\epsilon)$-approximation to the maximum matching. This chapter is based mostly on [68]. We note that the proofs here are considerably shorter than those of [68], as we use the newer techniques of Chapter 4. We also note that the results of this chapter have been improved upon: Even et al. [29] give a deterministic LCA that computes a maximal matching and requires $O(\log^* n)$ probes.

5.1 Preliminaries

For an undirected graph $G = (V, E)$, a *matching* is a subset of edges $M \subseteq E$ such that no two edges $e_1, e_2 \in M$ share a vertex. We denote by M^* a matching of maximum cardinality. An *augmenting path* with respect to a matching M is a simple path whose endpoints are *free* (i.e., not part of any edge in the matching M), and whose edges alternate between $E \setminus M$ and M. A set of augmenting paths P is *independent* if no two paths $p_1, p_2 \in P$ share a vertex.

For sets A and B, we denote $A \oplus B \stackrel{def}{=} (A \cup B) \setminus (A \cap B)$. An important observation regarding augmenting paths and matchings is the following.

Observation 5.1.1. *If M is a matching and P is an independent set of augmenting paths, then $M \oplus P$ is a matching of size $|M| + |P|$.*

In the distributed setting, Itai and Israeli [48] showed a randomized algorithm that computes a maximal matching (which is a 1/2-approximation to the maximum matching) and runs in $O(\log n)$ time with high probability. This result has been improved several times since (e.g., [24, 45]); of particular relevance is the approximation scheme of Lotker et al. [62], which, for every $\epsilon > 0$, computes a $(1-\epsilon)$-approximation to the maximum matching in $O(\log n)$ time. Kuhn et al. [55] proved that any distributed algorithm, randomized or deterministic, requires (in expectation) $\Omega(\sqrt{\log n / \log\log n})$ time to compute a $\Theta(1)$-approximation to the maximum matching, even if the message size is unbounded.

Our algorithm is, in essence, an implementation of the abstract algorithm of Lotker et al. [62]. Their algorithm, relies on several interesting results due to Hopcroft and Karp [46]. The algorithm of Even et al. [29] is a similar implementation of Hopcroft and Karp's algorithm; it has a probe complexity of $O(\log^* n)$, and is deterministic.

5.2 Distributed Maximal Matching

While the main result of Hopcroft and Karp [46] is an improved time complexity matching algorithm for bipartite graphs, they show the following useful lemmas for general graphs. The first lemma shows that if the current matching has augmenting paths of length at least ℓ, then using a maximal set of augmenting paths of length ℓ will result in a matching for which the shortest augmenting path is strictly longer than ℓ.

Lemma 5.2.1. *[46] Let $G = (V, E)$ be an undirected graph, and let M be some matching in G. If the shortest augmenting path with respect to M has length ℓ and Φ is a maximal set of independent augmenting paths of length ℓ, the shortest augmenting path with respect to $M \oplus \Phi$ has length strictly greater than ℓ.*

The second lemma shows that if there are no short augmenting paths then the current matching is approximately optimal.

Lemma 5.2.2. *[46] Let $G = (V, E)$ be an undirected graph. Let M be some matching in G, and let M^* be a maximum matching in G. If the shortest augmenting path with respect to M has length $2k - 1 > 1$ then $|M| \geq (1 - 1/k)|M^*|$.*

Lotker et al. [62] gave the following abstract approximation scheme for maximal

matching in the distributed setting.[1] Start with an empty matching. In stage $\ell = 1, 3, \ldots, 2k-1$, add a maximal independent collection of augmenting paths of length ℓ. For $k = \lceil 1/\epsilon \rceil$, by Lemma 5.2.2, we have that the matching M_ℓ is a $(1-\epsilon)$-approximation to the maximum matching.

In order to find such a collection of augmenting paths of length ℓ, we need to define a conflict graph:

Definition 5.2.3. *[62] Let $G = (V, E)$ be an undirected graph, let $M \subseteq E$ be a matching, and let $\ell > 0$ be an integer. The ℓ-conflict graph with respect to M in G, denoted $C_M(\ell)$, is defined as follows. The nodes of $C_M(\ell)$ are all augmenting paths of length ℓ, with respect to M, and two nodes in $C_M(\ell)$ are connected by an edge if and only if their corresponding augmenting paths intersect at a vertex of G.*[2]

We present the abstract distributed algorithm of [62], AbstractDistributedMM.

Algorithm 3: AbstractDistributedMM

Input : $G = (V, E)$ and $\epsilon > 0$
Output: A matching M

1 $M_{-1} \leftarrow \emptyset$; // M_{-1} is the empty matching
2 $k \leftarrow \lceil 1/\epsilon \rceil$;
3 **for** $\ell = 1, 3, \ldots, 2k-1$ **do**
4 Construct the conflict graph $C_{M_{\ell-2}}(\ell)$;
5 Let $\mathcal{I}$ be an MIS of $C_{M_{\ell-2}}(\ell)$;
6 Let $\Phi(M_{\ell-2})$ be the union of augmenting paths corresponding to $\mathcal{I}$;
7 $M_\ell \leftarrow M_{\ell-2} \oplus \Phi(M_{\ell-2})$; // M_ℓ is matching at the end of phase ℓ
8 Output M_ℓ; // M_ℓ is a $(1 - \frac{1}{k+1})$-approximate maximum matching

Note that for M_ℓ, the minimal augmenting path is of length at least $\ell + 2$. This follows since $\Phi(M_{\ell-2})$ is a maximal independent set of augmenting paths of length ℓ. When we add $\Phi(M_{\ell-2})$ to $M_{\ell-2}$, to get M_ℓ, by Lemma 5.2.1 all the remaining augmenting paths are of length at least $\ell + 2$ (recall that augmenting paths have odd lengths).

Lines 4 - 7 do the task of computing M_ℓ as follows: the conflict graph $C_{M_{\ell-2}}(\ell)$ is constructed and an MIS, $\Phi(M_{\ell-2})$, is found in it. $\Phi(M_{\ell-2})$ is then used to augment

[1] This approach was first used by Hopcroft and Karp in [46]; however, they only applied it efficiently in the bipartite setting.

[2] Notice that the nodes of the conflict graph represent *paths* in G. Although it should be clear from the context, in order to minimize confusion, we refer to a vertex in G by *vertex*, and to a vertex in the conflict graph by *node*.

$M_{\ell-2}$, to give M_ℓ.

We would like to simulate this algorithm locally. Our main challenge is to simulate Lines 4 - 7 without explicitly constructing the entire conflict graph $C_{M_{\ell-2}}(\ell)$. To do this, we will simulate the online greedy MIS algorithm. When simulating GreedyMIS on the conflict graph $C_M(\ell) = (V_{C_M}, E_{C_M})$, we only need a subset of the nodes, $V' \subseteq V_{C_M}$. Therefore, there is no need to construct $C_M(\ell)$ entirely; only the relevant subgraph need be constructed. By Theorem 4.5.2, we know there is an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA for MIS on the conflict graph.

5.2.1 LCA for Maximal Matching

Our main result for this chapter is the following theorem:

Theorem 5.2.4. *Let $G = (V, E)$ be a graph of bounded degree d. Then there exists an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA that, for every $\epsilon > 0$, computes a maximal matching that is a $(1-\epsilon)$-approximation to the maximum matching.*

We present our algorithm for maximal matching - LocalMM, and analyze it. The pseudocode for LocalMM (Algorithm 4) appears in Section 5.4. In contrast to the distributed algorithm, which runs iteratively, LocalMM is recursive in nature. In each iteration of the **for** in Algorithm AbstractDistributedMM, a maximal matching M_ℓ, is computed, where M_ℓ has no augmenting path of length less than ℓ. We call each such iteration a *phase*, and there are a total of k phases: $1, 3, \ldots 2k-1$. To find out whether an edge $e \in E$ is in M_ℓ, we recursively compute whether it is in $M_{\ell-2}$ and whether it is in $\Phi(M_{\ell-2})$, a maximal set of augmenting paths of length ℓ. We use the following simple observation to determine whether $e \in M_\ell$. The observation follows since $M_\ell \leftarrow M_{\ell-2} \oplus \Phi(M_{\ell-2})$.

Observation 5.2.5. *Edge $e \in M_\ell$ if and only if it is in either in $M_{\ell-2}$ or in $\Phi(M_{\ell-2})$, but not in both.*

Recall that LocalMM receives an edge $e \in E$ as a query, and outputs "yes/no". To determine whether $e \in M_{2k-1}$, it therefore suffices to determine, for $\ell = 1, 3, \ldots, 2k-3$, whether $e \in M_\ell$ and whether $e \in \Phi(M_\ell)$.

We will outline our algorithm by tracking a single query. (The initialization parameters will be explained at the end.) When queried on an edge e, LocalMM calls the procedure IsInMatching with e and the number of phases k. For clarity, we

sometimes omit some of the parameters from the descriptions of the procedures.

Procedure 6: IsInMatching determines whether an edge e is in the matching M_ℓ. To determine whether $e \in M_\ell$, IsInMatching recursively checks whether $e \in M_{\ell-2}$, by calling IsInMatching$(\ell-2)$, and whether e is in some path in the MIS $\Phi(M_{\ell-2})$ of $C_{M_{\ell-2}}(\ell)$. This is done by generating all paths p of length ℓ that include e, and calling IsPathInMIS(p) on each. IsPathInMIS(p) checks whether p is an augmenting path, and if so, whether it in the independent set of augmenting paths. By Observation 5.2.5, we can compute whether e is in M_ℓ given the output of the calls.

Procedure 7: IsPathInMIS receives a path p and returns whether the path is in the MIS of augmenting paths of length ℓ. The procedure first computes all the relevant augmenting paths (relative to p) using RelevantPaths (the relevant augmenting paths are all the paths of length ℓ that our LCA needs to consider in order to decide whether p is in the MIS). Given the set of relevant paths (represented by nodes) and the intersection between them (represented by edges) we simulate GreedyMIS on this subgraph. The resulting independent set is a set of independent augmenting paths. We then just need to check if the path p is in that set.

Procedure 9: RelevantPaths receives a path p and returns all the relevant augmenting paths relative to p. The procedure returns the subgraph of $C_{M_{\ell-2}}(\ell), C = (V_C, E_C)$, which includes p and all the relevant nodes. These are exactly the nodes needed for the simulation of GreedyMIS, given the order induced by seed s_ℓ. The set of augmenting paths V_C is constructed iteratively, by adding an augmenting path q if it intersects some path $q' \in V_C$ and arrives before it (i.e., $r(q, s_\ell) < r(q', s_\ell)$). In order to determine whether to add path q to V_C, we need first to test if q is indeed a valid augmenting path, which is done using IsAnAugmentingPath.

Procedure 10: IsAnAugmentingPath tests if a given path p is an augmenting path. It is based on the observation that p is an augmenting path with respect to a matching M if and only if all odd numbered edges are not in M, all even numbered

edges are in M, and both the vertices at the ends of p are free.

Given a path p of length ℓ, to determine whether $p \in C_{M_{\ell-2}}(\ell)$, IsAnAugmentingPath($\ell$) determines, for each edge in the path, whether it is in $M_{\ell-2}$, by calling IsInMatching($\ell-2$). It also checks whether the end vertices are free, by calling Procedure IsFree(ℓ), which checks, for each vertex, if any of its adjacent edges are in $M_{\ell-2}$.Thus, IsAnAugmentingPath(ℓ) correctly determines whether p is an augmenting with respect to $M_{\ell-2}$.

We end by describing the initialization procedure Initialize, which is run only once, during the first query. The procedure sets the number of phases to $\lceil 1/\epsilon \rceil$. It is important to set a different seed s_ℓ for each phase ℓ, since the conflict graphs are unrelated (and even the size of the description of each node, a path of length ℓ, is different). The lengths of the k seeds, $s_1, s_3, \ldots, s_{2k-1}$, determine our memory requirement.

5.3 Bounding the Complexity

In this section we prove Theorem 5.2.4. We start with the following observation:

Observation 5.3.1. *In any graph $G = (V, E)$ with bounded degree d, each edge $e \in E$ can be part of at most $\ell(d-1)^{\ell-1}$ paths of length ℓ. Furthermore, given e, it takes at most $O(\ell(d-1)^{\ell-1})$ time to find all such paths.*

Remark 5.3.2. *This observation does not hold for d-light graphs. If there are q vertices of degree $\log n$ that are close to each other in the graph, their adjacent edges can be part of $\log^q n$ paths. It is easy to verify, though, that the algorithm is a $(O(\text{polylog}\, n), O(\text{polylog}\, n), O(\text{polylog}\, n), O(\text{polylog}\, n), 1/n)$-LCA on d-light graphs.*

Proof of Observation 5.3.1. Consider a path $p = (e_1, e_2, \ldots, e_\ell)$ of length ℓ. If p includes the edge e, then e can be in one of the ℓ positions. Given that $e_i = e$, there are at most $d-1$ possibilities for e_{i+1} and for e_{i-1}, which implies at most $(d-1)^{\ell-1}$ possibilities to complete the path to be of length ℓ. □

Observation 5.3.1 yields the following corollary.

Corollary 5.3.3. *The ℓ-conflict graph with respect to any matching M in $G = (V, E)$, $C_M(\ell)$, consists of at most $\ell(d-1)^{\ell-1}|E| = O(|V|)$ nodes, and has maximal degree at most $d(\ell+1)\ell(d-1)^{\ell-1}$.*

Proof. (For the degree bound.) Each path has length ℓ, and therefore has $\ell+1$ vertices. Each vertex has degree at most d, which implies $d(\ell+1)$ edges. Each edge is in at most $\ell(d-1)^{\ell-1}$ paths. □

Our main task will be to compute a bound on the number of recursive calls. First, let us summarize a recursive call. The only procedure whose runtime depends on the order induced by s_ℓ is RelevantPaths, which depends on the number of vertices V_C (which is a random variable depending of the seed s_ℓ). To simplify the notation we let $\xi_\ell = d(\ell+1)\ell(d-1)^{\ell-1}$ and define the random variable $X_\ell = \xi_\ell |V_C|$. Technically, GreedyMIS also depends on V_C, but its running time is dominated by the running time of RelevantPaths.

Calling procedure	Called Procedures
IsInMatching(ℓ)	$1\times$ IsInMatching($\ell-2$) and $\ell(d-1)^{\ell-1}\times$ IsPathInMIS(ℓ)
IsPathInMIS(ℓ)	$1\times$ RelevantPaths(ℓ) and $1\times$ GreedyMIS
RelevantPaths(ℓ)	$X_\ell\times$ IsAnAugmentingPath(ℓ)
IsAnAugmentingPath(ℓ)	$\ell\times$ IsInMatching($\ell-2$) and $2\times$ IsFree(ℓ)
IsFree(ℓ)	$(d-1)\times$ IsInMatching($\ell-2$)

Table 5.1: Number of calls from each procedure

From the table, it is easy to deduce the following proposition.

Proposition 5.3.4. IsAnAugmentingPath(ℓ) *generates at most* $\ell+2(d-1)$ *calls to* IsInMatching($\ell-2$)*, and therefore at most* $(\ell+2d-2)\cdot\ell(d-1)^{\ell-1}$ *calls to* IsPathInMIS($\ell-2$)*.*

We would like to bound X_ℓ, the number of calls to IsAnAugmentingPath(ℓ) during a single execution of IsPathInMIS(G,p,ℓ,S). $|V_C|$ is exactly the size of the relevant vicinity of p in $C_{M_{\ell-2}}(\ell)$.

Denote by f_ℓ the number of calls to IsAnAugmentingPath(ℓ) during one execution of LocalMM. Let $f=\sum_{\ell=1}^{2k-1} f_\ell$.[3] The base cases of the recursive calls LocalMM makes are IsAnAugmentingPath(1) (which always returns TRUE). As the execution of each procedure of LocalMM results in at least one call to IsAnAugmentingPath,

[3]For all even ℓ, let $f_\ell = 0$.

f (multiplied by some small constant) is an upper bound to the total number of computations made by LocalMM.

We prove the following:

Proposition 5.3.5. *For every $1 \leq \ell \leq 2k-1$, there exist a constant c_ℓ, which depends only on d, such that*

$$Pr[f_\ell > c_\ell \log n] \leq \frac{2k-\ell}{n^4}.$$

Proof. The proof is by induction. For the base of the induction, we have, from Corollary 4.3.6, that there exists an absolute constant c_{2k-1}, which depends only on d, such that

$$\Pr[f_{2k-1} > c_{2k-1} \log n] \leq \frac{1}{n^4}.$$

For the inductive step, we use the law of total probability:

$$\begin{aligned}
\Pr[f_i > c_i \log n] =& [f_i > c_i \log n | f_{i+1} \leq c_{i+1} \log n] \Pr[f_{i+1} \leq c_{i+1} \log n] \\
&+ [f_i > c_i \log n | f_{i+1} > c_{i+1} \log n] \Pr[f_{i+1} > c_{i+1} \log n] \\
\leq& [f_i > c_i \log n | f_{i+1} \leq c_{i+1} \log n] + \Pr[f_{i+1} > c_{i+1} \log n] \\
\leq& \frac{1}{n^4} + \frac{2k-i-1}{n^4} \qquad (5.1) \\
=& \frac{2k-i}{n^4}
\end{aligned}$$

where Inequality (5.1) is due to Corollary 4.3.8 and the inductive hypothesis. □

Taking a union bound over all k levels immediately gives

Lemma 5.3.6. *There exists a constant c, which depends only on d and ϵ, such that*

$$\Pr[f > c \log n] \leq \frac{1}{n^3}.$$

Proof of Theorem 5.2.4. Using Lemma 5.3.6, and taking a union bound over all possible queried edges gives us that with probability at least $1 - 1/n$, LocalMM will require at most $O(\log n)$ probes to G. Therefore, for each execution of LocalMM, we require at most $O(\log n)$-independence for each conflict graph, and therefore, from Theorem 4.1.8, we require $\lceil 1/\epsilon \rceil$ seeds of length $O(\log n)$, which upper bounds the space required by the algorithm. The time required is upper bound by the time required to compute

$r(p)$ for all the required nodes in the conflict graphs, which is $O(\log^2 n)$. □

5.4 Pseudocode

Algorithm 4: LocalMM

Input : $G = (V, E)$, $e \in E$ and $\epsilon > 0$
Output: yes/no

Seed $\mathcal{S}$;
if *this is the first execution of* LocalMM **then**
  $(\mathcal{S}, k) \leftarrow$ Initialize(G, ϵ) ;
Return IsInMatching$(G, e, 2k-1, \mathcal{S})$.

Procedure 5: Initialize$(G = (V, E), \epsilon)$

`// This is run only at the first execution`

$n \leftarrow |V|$;
$k \leftarrow \lceil 1/\epsilon \rceil$;
for $\ell \leftarrow 1, 3, \ldots 2k-1$ **do**
  Generate an almost $O(\log n)$-wise independent seed $\mathcal{S}_\ell$ of length $O(\log n)$;
$\mathcal{S} = \bigcup_\ell \mathcal{S}_\ell$;
Return $(\mathcal{S}, k)$;

Procedure 6: IsInMatching$(G,\ e,\ \ell,\ \mathcal{S})$

if $\ell = -1$ **then** `// The empty matching`
  Return false
$b_1 \leftarrow$ IsInMatching$(G,\ e,\ \ell - 2,\ \mathcal{S})$;
$b_2 \leftarrow$ false ;
$P \leftarrow \{p \in G : e \in p \wedge |p| = \ell\}$;
for *all* $p \in P$ **do**
  if IsPathInMIS$(G,\ p,\ \ell,\ \mathcal{S})$ **then**
    $b_2 \leftarrow$ true ;
Return $b_1 \oplus b_2$;

Procedure 7: IsPathInMIS(G, p, ℓ, $\mathcal{S}$)

$C \leftarrow$ RelevantPaths(G, p, ℓ, $\mathcal{S}$) ; // C is a subgraph of $C_{M_{\ell-2}}(\ell)$
$I \leftarrow$ Greedy MIS $(C, \pi(C, s_\ell))$;
Return $(v \in I)$;

Procedure 8: IsFree(G, v, ℓ, $\mathcal{S}$)

IsFreeVertex $\leftarrow$ true ;
for *all* $u \in N(v)$ **do** // All edges touching v
  if IsInMatching$(G, (u, v), \ell - 2, \mathcal{S})$ **then**
    IsFreeVertex $\leftarrow$ false ;
Return IsFreeVertex ;

Procedure 9: RelevantPaths(G, p, ℓ, $\mathcal{S}$)

Initialize $C = (V_C, E_C) \leftarrow (\emptyset, \emptyset)$;
if IsAnAugmentingPath$(G, p, \ell, \mathcal{S})$ **then**
  $V_C \leftarrow \{p\}$;
else
  Return C ;
while $\exists p \in V_C : (p, p') \in E_C, r_\ell(p', s_\ell) < r_\ell(p, s_\ell)$ **do**
  if IsAnAugmentingPath$(G, p', \ell, \mathcal{S})$ **then**
    $V_C \leftarrow p'$;
    for *all* $p'' \in N(p')$ **do** // Edges between p' and vertices in V_C
      if $p'' \in V_C$ **then**
        $E_C \leftarrow (p', p'')$;
Return C ;

Procedure 10: IsAnAugmentingPath(G, p, ℓ, $\mathcal{S}$)

```
// Checks that p is an augmenting path.
if ℓ = 1 then // all edges are augmenting paths of the empty matching
    Return TRUE ;
Let p ← (e_1, e_2, ..., e_ℓ), with end vertices v_1, v_{ℓ+1} ;
IsPath ← true ;
for i = 1 to ℓ do
    if i (mod 2) = 0 then   // All even numbered edges should be in the
    matching
        if ¬IsInMatching(G, e_i, ℓ − 2, S) then
            IsPath ← false ;
    if i (mod 2) = 1 then     // No odd numbered edges should be in the
    matching
        if IsInMatching(G, e_i, ℓ − 2, S) then
            IsPath ← false ;
if (¬IsFree(G, v_1, ℓ, S) ∨ ¬IsFree(G, v_{ℓ+1}, ℓ, S)) then
    IsPath ← false ;          // The vertices at the end should be free
Return IsPath ;
```

Chapter 6

Constant-Time LCAs

In this chapter we give constant-time, constant-probe LCAs to the following graph problems, assuming graphs with constant maximal degree.

- *Graphic Matroids.* Given a weighted graph, the task is to find an acyclic edge set (forest) of approximately the maximum possible weight. In the corresponding LCA, a query specifies an edge, and the algorithm says whether the given edge is in the solution forest. We present a deterministic $(1-\epsilon)$-approximate LCA for graphic matroids, whose running time and space are independent of the size of the matroid (see Section 4.1 for formal definitions).

- *Integer Multi-Commodity Flow (IMCF) and Multicut on Trees.* Given a tree with capacitated edges and source-destination pairs, the goal of IMCF is to route the greatest possible total flow where each pair represents a different commodity, subject to edge capacity constraints. Multicut is the dual problem where the goal is to pick an edge set of minimal total capacity so that no source can be connected to its destination without using a selected edge. We give a deterministic LCA for IMCF and multicut on trees that runs in constant time and gives a (1/4)-approximation to the optimal IMCF and a 4-approximation to the minimum multicut. We also give a randomized LCA to IMCF, with constant running time, very little enduring memory (less than a word), and expected approximation ratio $\frac{1}{2}-\epsilon$ for any constant $\epsilon > 0$.

- *Weighted Matching.* Given a weighted graph, we would like to approximate the maximum weight matching. We design a deterministic reduction from any (possibly randomized) LCA A for unweighted matching with approximation ratio α

to weighted matching with approximation ratio $\alpha/8$. Our reduction invokes A a constant number of times. Both the running time and approximation ratio are independent of the magnitude of the edge weights. We also give a constant-time 1/2 - approximation algorithm to maximum cardinality matching, however its enduring memory requirement is $O(\log n)$. Combining these results gives a constant-time 1/16 - approximation algorithm to maximum weighted matching. It is not currently known whether there exists an approximation algorithm to maximum matching that requires constant time and memory. It is interesting to compare these results to those of Even et al. [29]. They give a $(1-\epsilon)$ approximation LCA to weighted matchings on graphs of constant bounded degree; its probe complexity is $O(\log^* n \log(w_{max} - w_{min}))$, where w_{max} and w_{min} are the weights of the heaviest and lightest edges respectively (or $O(\log^* n \log n)$ if $w_{max} - w_{min} > n$). Our reduction gives a worse approximation ratio, but removes the dependence on n of the running time.

This chapter is based on [66].

6.1 Graphic Matroids

Definition 6.1.1. *A matroid $\mathcal{M} = (E, \mathcal{I})$ is an ordered pair, where E is a finite set of elements (called the* ground set*), and $\mathcal{I}$ is a family of subsets of E, (called the* independent sets*), which satisfies the following properties:*

1. $\emptyset \in \mathcal{I}$,
2. *If $X \in \mathcal{I}$ and $Y \subseteq X$ then $Y \in \mathcal{I}$,*
3. *If $X, Y \in \mathcal{I}$ and $|Y| < |X|$ then there is an element $e \in X$ such that $Y \cup \{e\} \in \mathcal{I}$.*

For a more comprehensive introduction to matroids, we refer the reader to [84].

In this section we consider the problem of finding the maximum weight basis of a *graphic matroid*, defined as follows.

Definition 6.1.2 (Graphic matroid)**.** *A* graphic matroid *is a matroid whose independent sets are forests in an undirected graph.*

We are given a graphic matroid $\mathcal{M} = (\mathcal{E}, \mathcal{I})$, and would like to find an independent set of (approximately) maximal weight. In graph terminology, we are given a graph

$G = (V, E)$, with non-negative edge weights, $w : E \to \mathbb{R}^+$; we would like to find an acyclic set of edges of (approximately) maximal weight. That is, we seek a forest whose weight is close to the weight of a maximal spanning tree (MaxST). Without loss of generality, we assume edge weights are distinct, as it is always possible to break ties by ID. Recall that when the edge weights are distinct, there is a unique MaxST.

We first describe a parallel algorithm for finding a MaxST in a graph, and then explain how to adapt it to an LCA. Our parallel algorithm is less efficient than others (say, the Borůvka's algorithm [77]), but adapting it to an LCA is easy, and proving it correct, or analyzing its local properties, are simple.

We first need a few definitions. Define the distance between a vertex v and an edge $e = (u, w)$, denoted $\text{dist}(v, e)$, to be $\min\{\text{dist}(v, u), \text{dist}(v, w)\}$.

Definition 6.1.3 (Connected component, Truncated CC)**.** *Let $G = (V, E)$ be a simple undirected graph. For a vertex $v \in V$, and a subset of edges $\mathcal{S} \subseteq E$, the connected component of v with respect to $\mathcal{S}$ is $\text{CC}_{\mathcal{S}}(v) \subseteq \mathcal{S}$ which includes the edges $e \in \mathcal{S}$ that have a path from e to v using only edges in $\mathcal{S}$. (Note that $\text{CC}_{\mathcal{S}}(\cdot)$ induces a partition of $\mathcal{S}$.) The k-*truncated connected component *of v is the set of all vertices in the connected component of v at distance at most k from v (w.r.t. G). We denote it by $k\,\text{TCC}_{\mathcal{S}}(v)$.*

Algorithm 11 works as follows. We maintain a forest $\mathcal{S}$, initially empty. For any vertex v, denote $\Gamma_v^k = k\,\text{TCC}_{\mathcal{S}}(v)$. In round k, vertex v considers the cut $(\Gamma_v^k, V \setminus \Gamma_v^k)$ and adds the heaviest edge of the cut, say e, to $\mathcal{S}$. (Note that k is both the round number and the radius parameter of the truncated connected component.) In contrast to many MaxST algorithms, an edge can be considered more than once, and it is possible that an edge e is considered—and even added—when it is already in $\mathcal{S}$ (if we add e to $\mathcal{S}$ when $e \in \mathcal{S}$, $\mathcal{S}$ remains the same).

6.1.1 Correctness of Algorithm 11

The correctness of Algorithm 11 relies on the so-called "blue rule" [103].

Lemma 6.1.4 ([103])**.** *Let C be any cut of the graph. Then the heaviest edge in C belongs to* MaxST*.*

Corollary 6.1.5. *All edges added to $\mathcal{S}$ by algorithm 11 are in* MaxST*.*

Corollary 6.1.5 establishes the correctness of Algorithm 11.

Algorithm 11: Parallel (CREW) MaxST Approximation Algorithm.

Input : $G = (V, E)$ with weight function $w : E \to \mathbb{R}^+, \epsilon > 0$
Output: a forest $\mathcal{S}$
`// assume all edge weights are distinct`

For all $v \in V$, $\mathcal{S}_v = \emptyset$;
for *round* $k = 0$ *to* $1/\epsilon$ **do**
  For each vertex v, let $\Gamma_v^k = k\,\mathrm{TCC}_{\mathcal{S}}(v)$;
  for *all vertices* $v \in V$ *in parallel* **do**
    if e *is the heaviest edge of the cut* $(\Gamma_v^k, V \setminus \Gamma_v^k)$ **then**
      $\mathcal{S}_v = \mathcal{S}_v \cup \{e\}$;
Return $\mathcal{S} = \bigcup_{v \in V} \mathcal{S}_v$.

6.1.2 Approximation Guarantee

We now turn to analyze the approximation ratio of Algorithm 11. To this end, define $\mathcal{S}_k$ to be the set of edges of MaxST that were added to $\mathcal{S}$ in rounds $1, 2, \ldots, k$ (implying that $\mathcal{S}_k \subseteq \mathcal{S}_{k+1}$). Let $\mathbb{R}_k = \mathrm{MaxST} \setminus \mathcal{S}_k$.

Consider the *component tree* of MaxST, defined as follows: the node set is $\{CC_{\mathcal{S}_k}(v) \mid v \in V\}$, and the edge set is $\{(CC_{\mathcal{S}_k}(v), CC_{\mathcal{S}_k}(u)) \mid (v, u) \in \mathbb{R}_k\}$. In words, there is a node in the component tree for each connected component of $\mathcal{S}_k$, and there is an edge in the component tree iff there is an edge in $\mathbb{R}_k$ connecting nodes in the corresponding components. We choose an arbitrary component as the root of the component tree, and direct all the edges towards it; this way, each edge $e \in \mathbb{R}_k$ is outgoing from exactly one connected component of $\mathcal{S}_k$. We denote this component by CC_e^k. Note that $CC_e^i \subseteq CC_e^j$ for $i < j$ because components only grow. For any set of edges S, let $w(S) = \sum_{e \in S} w(e)$.

The following proposition is the key to the analysis.

Proposition 6.1.6. *For any* $k \geq 1$, $\forall e \in \mathbb{R}_k$, $w(e) \leq \frac{w(CC_e^k)}{k}$.

Proof. If $\mathbb{R}_k$ is empty, the claim holds trivially. Let $e = (v, u)$ be any edge in $\mathbb{R}_k$ (directed from v to u). Edge e was not chosen by vertex v in rounds $1, \ldots, k$. For $i \in [k]$, let e_i be the edge chosen by v in round i. It suffices to show that (1) all edges e_i are heavier than e, i.e. $\forall i \in [k], w(e_i) > w(e)$, and that (2) the edges e_i are distinct, i.e., $\forall i, j \in [k], i \neq j \;\Rightarrow\; e_i \neq e_j$.

The proof of (1) is straightforward: e was in the cut $(\Gamma_v^i, V \setminus \Gamma_v^i)$ in all rounds $i \in [k]$, but it was never chosen. This must be because v chose a heavier edge in each round.

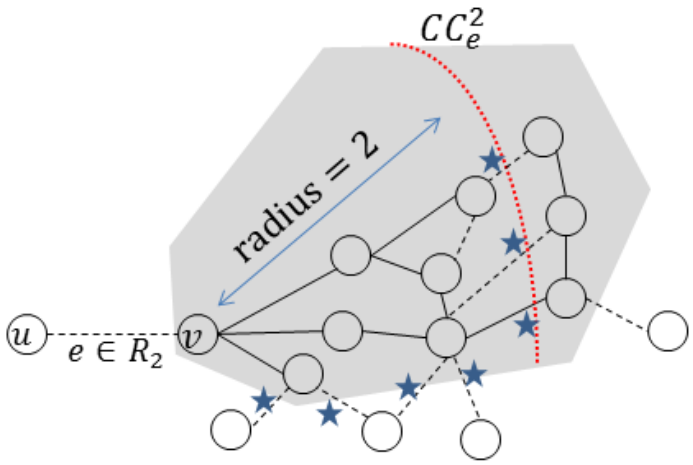

Figure 6.1: The situation considered in the proof of Proposition 6.1.6, for $k = 2$. The edge $e = (u, v)$ is in $\mathbb{R}_k$. Solid edges are in $\mathcal{S}$, dashed edges are in $E \setminus \mathcal{S}$. The shaded area represents CC_e^2, and the dotted arc represents the distance 2 range. Edges marked by $\star$ are considered by v in round 2.

To prove (2), we show by induction that e_i is distinct from $\{e_1, e_2, \ldots, e_{i-1}\}$. The base of the induction is trivial. For the inductive step, consider the two possible cases. If $e_i \notin CC_e^{i-1}$, then clearly, e_i cannot be any edge that was previously added. And if $e_i \in CC_e^{i-1}$, then e_i must be at distance exactly i from v, otherwise it would not have been in the cut $(\Gamma_v^i, V \setminus \Gamma_v^i)$ and could not have been added. But $e_1, \ldots, e_{i-1}$ are all at distance at most $i - 1$ (w.r.t. G). □

Corollary 6.1.7. *For $k \geq 0$, $w(\mathbb{R}_k) \leq \frac{w(\mathcal{S}_k)}{k}$.*

Proof.

$$
\begin{aligned}
w(\mathbb{R}_k) &= \sum_{e \in \mathbb{R}_k} w(e) \\
&\leq \sum_{e \in \mathbb{R}_k} \frac{w(CC_e^k)}{k} && \text{by Prop. 6.1.6} \\
&\leq \frac{w(\mathcal{S}_k)}{k} && e \neq e' \Rightarrow CC_e^k \neq CC_{e'}^k, \text{ and } \bigcup_{e \in \mathbb{R}_k} CC_e^k \subseteq \mathcal{S}_k
\end{aligned}
$$

□

This enables us to prove our approximation bound. Denote the weight of MaxST

by OPT.

Lemma 6.1.8. $w(\mathcal{S}_k) \geq (1 - \frac{1}{k+1})OPT$.

Proof. As $\mathbb{R}_k = \text{MaxST} \setminus \mathcal{S}_k$,

$$\frac{w(\mathcal{S}_k)}{\text{OPT}} = \frac{w(\mathcal{S}_k)}{w(\mathcal{S}_k) + w(\mathbb{R}_k)} \geq \frac{w(\mathcal{S}_k)}{w(\mathcal{S}_k) + w(\mathcal{S}_k)/k} = \frac{k}{k+1},$$

where the inequality is due to Corollary 6.1.7. □

This concludes the analysis of Algorithm 11. We now describe the LCA we derive from it.

6.1.3 Implementation and Complexity Analysis

Given a graph $G = (V, E)$ and a query $e = (u, v) \in E$, the implementation of Algorithm 11 as an LCA is as follows. Consider iteration k of Algorithm 11. Probe G to discover $N^{2k}(u)$ and $N^{2k}(v)$. Simulate Algorithm 11 on all vertices in $N^k(u) \cup N^k(v)$ for k rounds. In each round i, for each node $s \in N^k(u) \cup N^k(v)$, the algorithm computes $\Gamma_v^i = i\,\text{TCC}_{\mathcal{S}}(v)$, finds the heaviest edge e in the cut $(\Gamma_v^i, V \setminus \Gamma_v^i)$, and adds it to the solution. This gives the following lemma.

Lemma 6.1.9. *The time required to simulate the execution of Algorithm 11 for k rounds as an LCA is $kd^{O(k)}$, and the probe complexity is $d^{O(k)}$.*

Proof. The time to discover $N^{2k}(v) \cup N^{2k}(u)$ by probing the graph is bounded by $d^{O(k)}$. Each vertex in $z \in N^k(v) \cup N^k(u)$ executes Algorithm 11 k rounds: in round j, it constructs $\mathcal{S}_z$, by exploring $N^j(z) \subseteq N^{2k}(v) \cup N^{2k}(u)$. Overall, the time complexity is $d^{O(k)} + \sum_{z \in N^k(v) \cup N^k(u)} k|N^k(z)| = kd^{O(k)}$.

□

Combining Lemmas 6.1.8 and 6.1.9 gives the following result.

Theorem 6.1.10. *Let G be a graph whose degree is bounded by d. For every $\epsilon > 0$, there exists a deterministic $(d^{O(1/\epsilon)}, \frac{1}{\epsilon}d^{O(1/\epsilon)}, 0, \frac{1}{\epsilon}d^{O(1/\epsilon)}, 0)$-LCA, that computes a forest whose weight is a $(1-\epsilon)$-approximation to the maximal spanning tree of G.*

6.2 Multicut and Integer Multicommodity Flow in Trees

In this section we consider the integer multicommodity flow (IMCF) and multicut problems in trees. While simple, our LCAs demonstrate how, under some circumstances, one can find constant-time LCAs for apparently global problems.

The input is an undirected graph $G = (V, E)$ with a positive integer capacity $c(e)$ for each $e \in E$, and a set of pairs of vertices $\{(s_1, t_1), \ldots, (s_k, t_k)\}$. (The pairs are distinct, but the vertices are not necessarily distinct.)

In the *Integer Muticommdity Flow Problem*, the goal is to route commodity i from s_i to t_i so as to maximize the sum of the commodities routed, subject to edge capacity constraints. Note that in a tree, the only question is how much to route: the route is uniquely determined anyway. In the dual *Multicut Problem*, the goal is to find a minimum capacity *multicut*, where a multicut is an edge set that separates s_i from t_i for all $1 \leq i \leq k$.

We make the following assumptions about the input to allow for appropriately bound the time and space complexity of the algorithms. First, we assume that in the given tree each node has at most $d = O(1)$ children, and that T is rooted in the sense that the depth of each vertex is known and is part of the properties that are found by querying the vertex. Second, we assume that the distances from s_i to t_i are bounded by some given parameter ℓ; i.e., $\forall i, \text{dist}(s_i, t_i) \leq \ell$. Our bounds will be a function of ℓ, so that if ℓ is independent of tree size, then so are the time and space complexity of our algorithms.

As before, we adapt a classical algorithm to an LCA. This time we use the algorithm of Garg et al. [36] (Algorithm 12) as a subroutine.

Algorithm 12: Multicut and IMCF in trees [36, 107]

Input : A rooted tree T
Output: a flow f and a cut D

Initialize $f = 0, D = \emptyset$;
for *each vertex* v *in nonincreasing order of depth* **do**
  for *each pair* (s_i, t_i) *s.t. lowest common ancestor*$(s_i, t_i) = v$, **do**
    Greedily route flow from s_i to t_i if possible;
    Add all saturated edges to D in arbitrary order;
Let $e_1, \ldots, e_k$ be the ordered list of edges in D;
for $j = k$ *down to* 1 **do**
  If $D - \{e_j\}$ is a multicut, then remove e_j from D.;

Theorem 6.2.1 ([107]). *Algorithm 12 achieves approximation factors of* 2 *for the multicut problem and* 1/2 *for the IMCF problem on trees.*

Our deterministic LCA is detailed in Algorithm 13. It finds a $(4+\epsilon)$-approximation to the multicut problem, and in trees with minimum capacity $c_{\min} \geq 2$, the same algorithm finds an IMCF with approximation factor $\frac{\lfloor c_{\min}/2 \rfloor}{2c_{\min}} \geq \frac{1}{6}$.[1] We also present a randomized Algorithm (Algorithm 14), that gives an approximation factor of $(\frac{1}{2} - \epsilon)$ to IMCF for any desired $\epsilon > 0$ (the running time depends on $1/\epsilon$). The algorithms are similar, in that they partition the tree to subtrees and apply Algorithm 12 to each subtree. The randomized algorithm requires a very small amount of enduring memory, namely $O(\log(\ell/\epsilon))$ bits.

6.2.1 Deterministic LCA

We first describe the deterministic LCA. An edge is said to be at depth z if it connects vertices at depths $z-1$ and z. The deterministic algorithm (Algorithm 13) for multicut is as follows. We consider two overlapping decompositions of the tree into subtrees of height 2ℓ: the first decomposition is obtained by removing all edges at depth $k\ell$ for *even* values of k, and the second is by removing all edges at depth $k\ell$ for *odd* values of k. Now, given an edge e, we run Algorithm 12 on the subtrees that contain e (there is at least one such tree and at most two). Let the output of the "even" instance be D^e, and the output of the "odd" instance be D^o. The output is $D^e \cup D^o$. Correctness follows from the fact that each (s_i, t_i) pair is completely contained in (at least) one of the subtrees by the assumption that $\text{dist}(s_i, t_i) \leq \ell$. Regarding the approximation ratio, recall that by Theorem 6.2.1, the capacity of each of D^e, D^o is no larger than twice the minimum multicut capacity, the capacity in the overall output is at most 4 times that of the minimal capacity multicut.

To obtain a feasible flow, we start by splitting the capacity of each edge $e \in D$ between the subtrees it is a member of, such that in each subtree there are at least $\lfloor \frac{c_e}{2} \rfloor$ capacity units. This is possible because each edge is a member in at most 2 subtrees. Now, given a query e to the LCA, we run Algorithm 12 on the trees e is a member of, obtaining flow values $\vec{f}^e(e)$ for the "even" subtree and $\vec{f}^o(e)$ for the "odd" subtree. The LCA outputs $\vec{f}^e(e) + \vec{f}^o(e)$.

[1] The ratio is $\frac{1}{4}$ when all capacities are even, and it tends to $\frac{1}{4}$ as $c_{\min} \to \infty$. For $c_{\min} = 1$ the approximation ratio is 0.

Algorithm 13: Deterministic LCA for Multicut in Trees

Input : A graph $G = (V, E)$, with $c : E \to \mathbb{Z}^+$, and $\ell > 0$

Output : A multicut D

Set $h = 2\ell$;

Step 1.

- Delete all edges at depth $k\ell$, for odd k;
- On each remaining subtree T_i, run Algorithm 12;
- Let D_i be the multicut returned on T^i;
- Let $D^o = \cup_i D_i$;

Step 2.

- Delete all edges at depth $k\ell$, for even k;
- On each remaining subtree T_i, run Algorithm 12;
- Let D_j be the multicut returned on T^j;
- Let $D^e = \cup_j D_j$;

Step 3.

- Let $D = D^o \cup D^e$;
- Return D.

6.2.1.1 Approximation Guarantee

Let us call the subtrees created in **Step 1** of Algorithm 13 *odd subtrees*, and the subtrees created in **Step 2** *even subtrees*.

Lemma 6.2.2. *Algorithm 13 outputs a multicut, whose capacity is at most* 4 *times the capacity of the minimum multicut of* T.

Proof. Each vertex and edge is contained in two subtrees, and as the subtrees are of depth 2ℓ, each path (s_i, t_i) must be fully contained within either an odd subtree or an even subtree (or both). Therefore, at least one edge in $D_1 \cup D_2$ is on the path between s_i and t_i: D is a multicut. From Theorem 6.2.1, D_1 and D_2 are 2-approximations to the minimal capacity multicut of T, hence their union is at most a 4-approximation. □

Lemma 6.2.3. *Algorithm 13 outputs a feasible flow, whose value is at least* $(\frac{1}{4} - \frac{1}{4c_{\min}})$ *fraction of the maximal multicommodity flow on* T.

Proof. Let f^o and f^e denote the flows computed on odd and even trees, respectively. Their union is feasible because each of them uses at most half of the capacity of each

edge. Moreover, f_i is a 1/2 approximation to the flow of T_i. Denote the value of the maximal IMCF on T by f^*. Let f_i^* be the optimal flow on the subtree T_i. As in Algorithm 13, we denote indices of odd subtrees by i, and even subtrees by j. Then

$$\begin{aligned} f^* &\leq \sum_i f_i^* + \sum_j f_j^* \\ &\leq \sum_i 2f_i + \sum_j 2f_j \\ &\leq (2f^e + 2f^o) \cdot \frac{c_{\min}}{\lfloor c_{\min}/2 \rfloor} \\ &\leq \frac{4c_{\min}}{c_{\min} - 1}(f^e + f^o) \ . \end{aligned}$$

□

6.2.1.2 Complexity Analysis

The implementation of Algorithm 13 as an LCA is straightforward: when queried on an edge, determine to which subtrees it belongs, and then run simulate Algorithm 13 on those two subtrees. The maximal size of a subtree is $d^{2\ell}$, hence our probe complexity is $O(d^{2\ell})$. We do not attempt to optimize the running time; we note that it is trivial to implement the algorithm on subtree T_j in time $O(|T_j|^3)$, as there can be at most $|T_j|^2$ pairs (s_i, t_i) in T_j. This gives

Theorem 6.2.4. *Given a tree T with maximal degree d, integer edge capacities at least $c_{\min}$ and source-destination pairs with maximal distance at most $\ell > 0$, there is a deterministic $(d^{O(\ell)}, d^{O(\ell)}, 0, d^{O(\ell)}, 0)$-LCA for 4-approximate multicut and $(\frac{1}{4} - \frac{1}{4c_{\min}})$-approximate IMCF. If all capacities are even, the approximation ratio to IMCF is $\frac{1}{4}$.*

6.2.2 Randomized LCA

We now turn to the randomized setting. Our randomized algorithm (Algorithm 14) is very similar: instead of an overlapping decomposition, we use a random one as follows. Let $H = \lceil \frac{\ell}{\epsilon} \rceil$. We pick an integer j uniformly at random from $[H]$, and remove all edges whose depth modulo H is $j - 1$. The result is a collection of subtrees of depth at most $H - 1$ each. Now, given an edge e, we run Algorithm 12 on the subtree that contains e and output the output of Algorithm 12 (with probability $1/H$, the edge queried, e, is not in any tree; in this case, e carries 0 flow).

Algorithm 14: Integer Multicommodity Flow in Trees

Input : A graph $G = (V, E)$, with $c : E \to \mathbb{Z}^+$, and $\ell > 0$
Output : A flow f

Let $H = \lceil \frac{\ell}{\epsilon} \rceil$;
Uniformly sample an integer j from $\{0, \ldots, H\}$;
Delete all edges at depth $j + kH$, for all $k \in \mathbb{N}$ such that $j + kh \leq \text{depth}(T)$;
On each remaining subtree, run Algorithm 12;
Return the union of the flows on all subtrees.

Theorem 6.2.5. *Algorithm 14 achieves an approximation ratio of* $1/2 - \epsilon$ *to the maximum integer multicommodity flow on trees.*

Proof. For any i, the probability that the path (s_i, t_i) is not fully contained within a subtree is at most $\frac{\ell}{H} \leq \epsilon$. Fix an optimal solution f^* with value $|f^*|$. After deleting edges, the expected amount of remaining flow is at least $(1 - \epsilon)|f^*|$. By Theorem 6.2.1, Algorithm 12 output at least a half of that amount. The result follows. □

The implementation of Algorithm 14 as an LCA is almost identical to that of Algorithm 13; in order to achieve a $1/2 - \epsilon$ approximation to the IMCF, the enduring memory has to hold an integer whose value is at most $\lceil \frac{\ell}{\epsilon} \rceil$, i.e., $O(\log(\ell/\epsilon))$ bits. We therefore have

Theorem 6.2.6. *Given a tree T with maximal degree d, integer edge capacities and vertex pairs with maximal distance at most $\ell > 0$, there is a randomized $(d^{O(\ell/\epsilon)}, d^{O(\ell/\epsilon)}, 0, d^{O(\ell/\epsilon)}, 0)$-LCA that achieves an approximation ratio of $(1/2 - \epsilon)$ to IMCF.*

6.3 Weighted Matchings

In this section we present a different kind of an LCA: a reduction. Specifically, we consider the task of computing a maximum weight matching (MWM), and show how to *locally* reduce it to maximum cardinality matching (MCM). Our construction, given any graph of maximal degree d and a t-time α-approximation LCA for MCM, yields an $O(td)$-time, $\frac{\alpha}{8}$-approximation LCA for MWM.

Formally, in MWM we are given a graph $G = (V, E)$ with a weight function $w : E \to \mathbb{N}$, and we need to output a set of disjoint edges of (approximately) maximum total *weight*. In MCM, the task is to find a set of disjoint edges of (approximately) the largest possible *cardinality*.

The main idea in our reduction is a variant of the well-known technique of *scaling* (e.g., [63, 105, 109]): partition the edges into classes of more-or-less uniform weight, run an MCM instance for each class, and somehow combine the MCM outputs. Motivated by local computation, however, we use a very crude combining rule that lends itself naturally to LCAs.

Specifically, the algorithm is as follows (the "global" algorithm is presented as Algorithm 15). Let $\gamma = 4$. Partition the edges by weight to sets E_i, such that $E_i = \{e : w(e) \in [\gamma^{i-1}, \gamma^i)\}$. For each i, find a maximum cardinality matching M_i on the graph $G_i = (V, E_i)$, using any MCM algorithm. Let $M = \cup_i M_i$. Given an edge e, our LCA for MWM returns "yes" iff e is a local maximum in M, i.e., iff (1) e is in M, and (2) for any edge e' in M which shares a node with e, $w(e') < w(e)$ (no ties can occur).

Algorithm 15: Reduction of MWM to MCM

Input : A graph $G = (V, E)$, with $w : E \to \mathbb{N}$, and $\gamma > 2$
Output: A matching M

Partition the edges into *classes* $E_i = \{e : w(e) \in [\gamma^{i-1}, \gamma^i)\}$ for $i = 1, 2, \ldots$
In parallel, compute an unweighted matching M_i for each level i;
$M = \bigcup_i M_i$;
for *each edge* $e \in M$ **do**
  if *e has a neighbor $e' \in M$, with* $\text{class}(e') < \text{class}(e)$ **then**
    Remove e from M;
Return M.

Theorem 6.3.1. *Let $\mathcal{A}$ be a $(p(n), t(n), em(n), tm(n), \delta(n))$-LCA for unweighted matching, whose output is an α-approximation to the maximum matching. Then given a graph $G = (V, E)$ with maximal degree d and arbitrary weights on the edges, there is an $(O(d \cdot p(n)), O(d \cdot t(n)), O(d \cdot em(n)), O(d \cdot tm(n)), O(d \cdot \delta(n)))$-LCA that computes an $\alpha/8$-approximation to the maximum weighted matching.*

6.3.1 Correctness and Approximation Guarantee

No two adjacent edges within a class can be selected to M, as within each class we use a matching algorithm. Now consider two edges $e, e' \in M$ that are not in the same class (assume w.l.o.g. that $\text{class}(e') < \text{class}(e)$), and are adjacent in M after a matching has been computed for each class. Edge e is removed from M. Hence no adjacent edges can

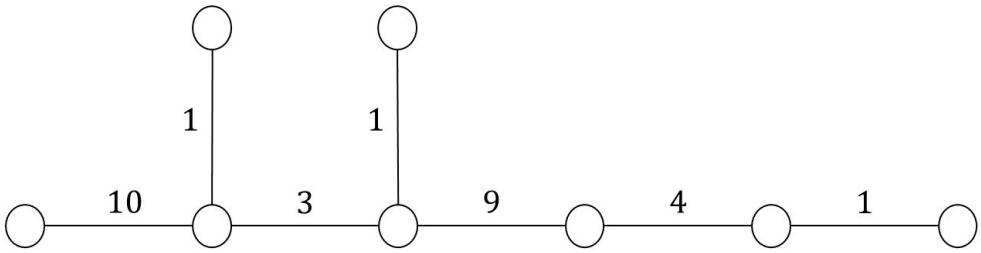

Figure 6.2: A simple example. Here $\gamma = 3$ and that $M = \cup_i M_i$ is the entire graph. The M-tree of the edge of weight 3 consists of the edge itself and its two neighbors of weight 1. The M-tree of the edge of weight 10 includes itself and the M-tree of the edge of weight 3. The M-tree of the edge of weight 9 is the entire graph, except the edge of weight 10.

be selected to M. The more interesting part is the approximation ratio analysis. For any edge $(u, v) = e \in M$, recursively define a tree T_e as follows. If e is lighter than all of its neighbors in M, then $T_e = \{e\}$. Otherwise, let f_u be the heaviest edge in M that touches u and is lighter than e. Define f_v similarly, and let $T_e = \{e\} \cup T_{f_u} \cup T_{f_v}$. (If f_u does not exist, let $T_{f_u} = \emptyset$; similarly f_v.) We call T_e the M-tree of e. See Figure 6.2 for an example.

Define a new weight function on G, $\hat{w}$: $\hat{w}(e) = \gamma^{\text{class}(e)-1}$; i.e., $\hat{w}(e)$ is $w(e)$ rounded down to the nearest power of γ. Note that the choices made by Algorithm 15 are identical under w and $\hat{w}$. For any set of edges S and weight function w, let $w(S) = \sum_{e \in S} w(e)$. The main argument in the analysis of the approximation ratio of Algorithm 15 is the following.

Proposition 6.3.2. *Let $e = (v, u)$ be any edge in M, such that $\hat{w}(e) = \gamma^k$ and let T_e be the M-tree of e. Then $\hat{w}(T_e) \leq \sum_{i=0}^{k} 2^{k-i}\gamma^i$.*

Proof. The proof is by induction on k. For the base of the induction, $k = 0$,we have $\hat{w}(T_e) = 2^0\gamma^0$. For the inductive step, assume that the proposition holds for all integers

up to $k-1$. Let $e=(u,v)$. The heaviest edge that is lighter than e and touches u (denoted f_u) weighs at most γ^{k-1}. Similarly for the heaviest edge that is lighter than e and touches v (f_v). Therefore, by the inductive hypothesis,

$$\hat{w}(T_e) \leq \gamma^k + 2\sum_{i=0}^{k-1} 2^{k-i-1}\gamma^i = \sum_{i=0}^{k} 2^{k-i}\gamma^i.$$

□

Corollary 6.3.3. *Let $e=(v,u)$ be any edge in M, such that $\hat{w}(e)=\gamma^k$ and let T_e be the M-tree of e. Then $\hat{w}(e) \geq \frac{\gamma-2}{\gamma}\hat{w}(T_e)$.*

Proof. As $\gamma > 2$,

$$\begin{aligned}
\hat{w}(T_e) &\leq \sum_{i=0}^{k} 2^{k-i}\gamma^i && \text{by Prop. 6.3.2} \\
&= \gamma^k \sum_{i=0}^{k} \frac{2^{k-i}}{\gamma^{k-i}} \\
&= \gamma^k \sum_{j=0}^{k} \frac{2^j}{\gamma^j} \\
&< \gamma^k \sum_{j=0}^{\infty} \frac{2^j}{\gamma^j} \\
&= \hat{w}(e)\frac{\gamma}{\gamma-2}.
\end{aligned}$$

□

Lemma 6.3.4. *Using any α-approximate MCM algorithm, Algorithm 15 finds a matching of total weight $\alpha\frac{\gamma-2}{\gamma^2}$OPT.*

Proof. Let M_i^* be a maximum weighted matching on $G_i=(V,E_i)$, and let $M^*=\cup_i M_i^*$. Let $\hat{M}_i^*$ be a maximum cardinality matching (MCM) on $\hat{G}_i$. Clearly, for all i, $w(\hat{M}_i^*) \geq \frac{1}{\gamma}w(M_i^*)$, because each edge in M_i^* weighs at most γ times any edge of $\hat{M}_i^*$, and M_i^* does not contain more edges than $\hat{M}_i^*$. Also note that $w(M^*) \geq$ OPT, because any restriction of an optimal MWM to edges of class i cannot have more weight than M_i^*. Call a locally heaviest edge in M an *output edge*. Note that every edge $e \in M$ is

contained in the M-tree of at least one output edge. We can therefore conclude that

$$\begin{aligned}
\sum_{e:e \text{ is an output edge}} w(e) &\geq \sum_{T_e:e \text{ is an output edge}} \frac{\gamma-2}{\gamma} w(T_e) && \text{by Corr. 6.3.3} \\
&\geq \frac{\gamma-2}{\gamma} w(M) \\
&= \frac{\gamma-2}{\gamma} \sum_i w(M_i) \\
&\geq \frac{\gamma-2}{\gamma} \sum_i \alpha w(\hat{M}_i^*) \\
&\geq \frac{\gamma-2}{\gamma^2} \sum_i \alpha w(M_i^*) \\
&= \alpha \frac{\gamma-2}{\gamma^2} w(M^*) \\
&\geq \alpha \frac{\gamma-2}{\gamma^2} \text{OPT}.
\end{aligned}$$

□

It is easy to verify that the optimal value of γ is 4, yielding approximation factor $\alpha/8$.

6.3.2 Complexity Analysis

The simulation of this algorithm as an LCA is simple, and, unlike the global algorithm, its complexity is completely independent of the weights on the edges. Suppose we are queried about edge e. Let N be the set of edges that includes e and all of its neighboring edges whose weight is greater than $w(e)$. We invoke, for each $e' \in N$, the LCA for MCM on the edges whose weight class is class(e'). The answer for e is "yes" iff the MCM LCA replied "yes" for the query on e, and "no" for all other queries. Theorem 6.3.1 follows.

6.3.3 Unweighted Matching

In order to guarantee that Algorithm 15 runs in constant time, we need to supply it with a constant-time LCA for unweighted matching. Unfortunately, no deterministic constant-time constant approximation algorithm exists for MCM [59]. There are, however, several constant-time approximation algorithms to MCM that one may use; for example, the constant-time MCM of Nguyen and Onak [79], or a straightforward implementation of Itai and Israeli's parallel MCM algorithm [48] as an LCA. We note,

however, that in both cases, we need enduring memory of size $O(\log n)$, as we need at least pairwise independence.

For completeness, we give the implementation of Itai and Israeli's algorithm, with a simple analysis. It is important to note that their algorithm finds an MCM and runs in $O(\log n)$ rounds. Using the reduction of Parnas and Ron (see Chapter 3) naively would yield an "LCA" with polynomial running time. We therefore have to terminate the simulation after a constant number of rounds. We show that we can find an arbitrarily good approximation to the MCM, in expectation. We note that it is not known whether there exists a constant-time MCM-approximation LCA that requires a constant enduring memory.

The algorithm we present is possibly the simplest conceivable randomized algorithm for maximal matching. Once again, we present it as a parallel algorithm, and then show how to implement it as an LCA. For ease of analysis, when an edge is added to the matching, we do not remove it or its neighbors from the graph. It is easy to modify the algorithm to do so; this can only improve its approximation factor.

Algorithm 16: Parallel (CREW) Maximal Matching Algorithm.

Input : $G = (V, E), \epsilon > 0$
Output: a matching M

$M = \emptyset$;
for $k = 0$ *to* $d^2 \ln \frac{2}{\epsilon}$ **do**
    For each vertex v, select a neighbor u uniformly at random;
    for *each edge* $e = (u, v)$ *in parallel* **do**
        if u *and* v *selected each other* **and** $M \cup \{e\}$ *is a matching* **then**
            $M = M \cup \{e\}$;
Return M.

Theorem 6.3.5. *Given a graph G of bounded degree d, for any ϵ, there is a randomized $(d^{O(d^2 \ln 1/\epsilon)}, d^{O(d^2 \ln 1/\epsilon)}, O(\log n \log 1/\epsilon), d^{O(d^2 \ln 1/\epsilon)}, 0)$-LCA that gives a $(1/2 - \epsilon)$-approximation to the maximum matching.*

We first prove some lemmas.

Lemma 6.3.6. *The output of Algorithm 16 is a $(1 - \frac{\epsilon}{2})$ approximation to some maximal matching.*

Proof. At the conclusion of Algorithm 16, it outputs a legal matching: whenever an edge

e is considered for addition to M, it is only added if $M \cup \{e\}$ is a legal matching. Note that even though we do the additions in parallel, there is no danger of two neighboring edges being added concurrently due to the edge selection process, which guarantees no vertex has more than one neighboring edge selected at any time. Let M^* be a maximal matching obtained by taking the matching output by Algorithm 16 and greedily adding edges to it until no more edges can be added. Define an edge to be *eligible* if it is in $M^* \setminus M$.

The probability that an eligible edge e is not added to M in a round is $1 - \frac{1}{d^2}$. Setting $k = d^2 \ln 2/\epsilon$, the probability that e is not added to M in any of the k rounds is

$$\left(1 - \frac{1}{d^2}\right)^k \leq \left(\frac{1}{e}\right)^{\ln 2/\epsilon} = \frac{\epsilon}{2}.$$

□

Similarly to Algorithm 11, it is straightforward to implement Algorithm 16 as an LCA: given a query $e = (u, v)$, if we wish to simulate Algorithm 16 for k rounds, we probe G to obtain $N^k(e)$. To see whether e is added to M in round i, it suffices to determine whether any of its neighbors $e' \in N(e)$ were added to M in any round up to $i - 1$. In order to determine this, we need to check whether any of them were added to M in any round up to $i - 2$, and so on. Therefore, to determine whether e is added to M after k rounds, we need to simulate Algorithm 16 on $N^i(e)$ for $k - i$ rounds. Therefore, we obtain

Lemma 6.3.7. *The time to execute Algorithm 11 for $k = d^2 \ln \frac{2}{\epsilon}$ is $O(d^{2k})$.*

Proof. Denote by $T(k)$ the time it takes to execute Algorithm 16 for k rounds. We show that $T(k) \leq (d)^{2k}$, by induction on the number of rounds, k. Assume we are given a edge $e = (u, v)$ as our query. For $k = 0$, we need to check if it is chosen by both endpoints, $O(1)$. Assume that the lemma holds for $j \leq k - 1$. On round k, we need to check whether e is selected, and whether it can be added to M, for which we need to check whether any of e's neighbors were previously added to M. This takes time

$$T(k) = 2dT(k-1) + O(1) = 2d \cdot d^{2k-2} + O(1) \leq d^{2k},$$

for $d \geq 2$. □

As the replies to all queries need to be consistent with the same matching, we

require that the random bits used by each vertex remain the same each time the LCA is invoked. For this, we need a source of randomness. Each query to the LCA requires the generation of at most d^{2k} random numbers $i \in [d]$. From Theorem 4.1.8, we have that a seed of length $O(\log(n/\epsilon))$ suffices, as we only require d^{2k}-wise independence, for $O(\log(1/\epsilon))$ rounds. This, along with Lemmas 6.3.6 and 6.3.7, proves the theorem.

Part II

Local Computation Mechanism Design

Chapter 7

Local Computation Mechanisms

In the second part of the thesis, we consider local computation mechanisms. Similarly to the previous chapters, we are only interested in a small part of the solution at any given time, but in addition, we need to meet some game theoretic requirements. In this chapter, we consider a relatively simple problem - the housing allocation problem. There is a set of houses and a set of agents, and the agents that have some preference over the houses. We wish to find an allocation of houses to agents that satisfies some properties. Most importantly, we would like the allocation to be *stable* (there are no two agents that would like to switch houses), and *truthful* (agents have no incentive to lie about their preferences). We could use one of the matching algorithms that we described in previous chapters, but that would guarantee neither stability nor truthfulness. In this and the coming chapters we show how we can accommodate these game-theoretic requirements. Chapters 7 through 10 are based on [42].

7.1 Notation and Preliminaries

We use the standard notation of game theoretic mechanisms. There is a set I of n rational agents and a set J of m items. In some settings, e.g., the stable marriage setting, there are no objects, only rational agents. Each agent $i \in I$ has a *valuation function* v_i that maps subsets $S \subseteq J$ of the items to non-negative numbers. The *utilities* of the agents are quasi-linear, namely, when agent i receives subset S of items and pays p, her utility is $u_i(S, p) = v_i(S) - p$. Agents are *rational* in the sense that they select actions to maximize their utility. We would like to allocate items to agents (or possibly agents to other agents), in order to meet global goal, e.g., maximize the sum of the

valuations of allocated objects (see, e.g., [80]).

A *mechanism with payments* $\mathcal{M} = (\mathcal{A}, \mathcal{P})$ is composed of an allocation function $\mathcal{A}$, which allocates items to agents, and a payment scheme $\mathcal{P}$, which assigns each agent a payment. A mechanism without payments consists only of an allocation function. Agents report their bids to the mechanism. Given the bids $b = (b_1, \ldots, b_n)$, the mechanism allocates the item subset $\mathcal{A}_i(b) \subseteq J$ to agent i, and, if the mechanism is with payments, charges her $\mathcal{P}_i(b)$; the utility of agent i is $u_i(b) = v_i(\mathcal{A}_i(b)) - \mathcal{P}_i(b)$.

A randomized mechanism is *universally truthful* if for every agent i, for every random choice of the mechanism, reporting her true private valuation maximizes her utility. A randomized mechanism is *truthful in expectation*, if for every agent i, reporting her true private valuation maximizes her expected utility. That is, for all agents i, any bids b_{-i} and b_i, $\mathbb{E}[u_i(v_i, b_{-i})] \geq \mathbb{E}[u_i(b_i, b_{-i})]$.

We say that an allocation function $\mathcal{A}$ *admits* a truthful payment scheme if there exists a payment scheme $\mathcal{P}$ such that the mechanism $\mathcal{M} = (\mathcal{A}, \mathcal{P})$ is truthful.

A mechanism $\mathcal{M} = (\mathcal{A}, \mathcal{P})$ fulfills *voluntary participation* if, when an agent bids truthfully, her utility is always non-negative, regardless of the other agents' bids, i.e., for all agents i and bids b_{-i}, $u_i(v_i, b_{-i}) \geq 0$.

Definition 7.1.1 (Mechanisms without payments)**.** *We say that a mechanism $\mathcal{M}$ is a* $(p(n), t(n), em(n), tm(n), \delta(n))$-local computation mechanism *if its allocation function is computed by a* $(p(n), t(n), em(n), tm(n), \delta(n))$*-LCA.*

Definition 7.1.2 (Mechanisms with payments)**.** *We say that a mechanism $\mathcal{M} = (\mathcal{A}, \mathcal{P})$ is a* $(p(n), t(n), em(n), tm(n), \delta(n))$-local computation mechanism *if both the allocation function $\mathcal{A}$ and the payment scheme $\mathcal{P}$ are computed by* $(p(n), t(n), em(n), tm(n), \delta(n))$*-LCAs.*

In other words, given a query x, $\mathcal{A}$ computes an allocation and $\mathcal{P}$ computes a payment, and both are LCAs. Furthermore, the replies of $\mathcal{A}$ to all of the queries are consistent with a single feasible allocation.

A *truthful local mechanism* $\mathcal{M} = (\mathcal{A}, \mathcal{P})$ is a local mechanism that is also truthful. Namely, each agent's dominant bid is her true valuation, regardless of the fact that the mechanism is local.

7.2 Warm Up - Random Serial Dictatorship

We first show a very simple local computation mechanism for the housing problem. We would like to allocate n houses to n agents. Assume that each agent is interested in a constant number of houses, d, and that the preferences are drawn from the uniform distribution. Each agent i has a complete preference relation R_i over the d houses. In the random serial dictatorship (RSD) algorithm, a permutation over the agents is generated, and then each agent chooses her most preferred house out of the unallocated houses. This algorithm has some desirable properties: it is truthful, nonbossy[1] and neutral[2]; in fact, no other mechanism has all of these properties [101]. We note that the algorithm is a crisp, neighborhood-dependent online algorithm, and therefore can be implemented as an LCA: we can simply use the reduction of Chapter 4.

One other nice property of RSD is that each agent has the same probability of having the first choice. This is an intuitively necessary property for any algorithm to be considered "fair". We define this formally as follows.

Property 7.2.1. *Let $\mathcal{M}$ be a mechanism for the housing problem. Let X_i denote the event that agent i has the first choice under $\mathcal{M}$. If*

$$\forall_{i,j}, \Pr[X_i] - \Pr[X_j] < \epsilon,$$

we say that the mechanism is ϵ-fair.

Unfortunately, the reduction of Chapter 4 does not guarantee this property: an agent with a large ID has a much smaller probability of being given the first choice than one with a small ID. In fact, while the probability of the agent with the smallest ID being given the first choice is at least $\frac{1}{L}$ (where L is the range of the hash function), the agent with the largest ID has a vanishingly small probability of being given the first choice. In order to satisfy this notion of fairness, we can use a larger range for our hash function (at the expense of a larger seed). Instead of using the construction of Theorem 4.1.8, we can use Theorem 4.2.2, which immediately implies Property 7.2.1. We therefore have

[1] A mechanism is *nonbossy* if no agent can change the outcome of the mechanism without changing her own outcome.

[2] Informally, a mechanism is *neutral* if the outcome of the mechanism does not depend on the names of the goods. See [101] for a formal definition.

Theorem 7.2.2. *Let k be some constant integer $k > 0$. Consider a house allocation problem with n agents and n houses, and let each agent preference list length be bounded by k, where each list is drawn uniformly at random from the set of all possible lists of length k. Then, for any $\epsilon > 0$ there is an $(O(\log n), O(\log^2 n), O(\log^2 n), O(\log^2 n), 1/n)$ - local computation mechanism that is ϵ-fair and whose output is identical to the random serial dictatorship allocation algorithm that uses the same randomness.*

Chapter 8

Stable Matching

In the *stable matching problem*, we are given a set of men and a set of women. The men have preferences over the women and the women over the men. The goal is to compute a matching H that is stable; that is, there is no man and woman who prefer each other to their partner in H. Stable matching has been at the center of game-theoretic research since the seminal paper of Gale and Shapley [34] (see, e.g., [92] for an introduction and a summary of many important results). Roth and Rothblum [94] examined the scenario in which the preference lists are of bounded length; in most real-life scenarios, this is indeed the case. For example, a medical student will not submit a preference list for internship over all of the hospitals in the United States, but only a short list. We examine the variant in which each man $m \in M$ is interested in at most k women, (and prefers to be unmatched than to be matched to anyone not on their list; cf. [93]). We limit our attention to the setting in which the men's preferences are assumed to be uniformly distributed; cf. [47, 53].

The Gale-Shapley algorithm results in a stable matching regardless of the preferences (see, e.g., [95]); however, its running time is $\Omega(n^2)$. Indeed this is a lower bound on any algorithm that finds a stable matching (under full preference lists) [78]. Furthermore, it is known that a linear number of iterations of the Gale-Shapley algorithm is necessary to attain stability [39]. One direction taken to obtain sub-linear running time is executing parallel computation on instances with short preference lists. Feder et al. [31] proposed one such algorithm for stable matching. Unfortunately, it does not appear possible to convert their algorithm to an LCA, as they require m^4 processors, and the running time is $O(\sqrt{m}\log^3 n)$, where m is the sum of the preference list lengths. In some cases, matchings that are "almost" stable may be acceptable (see,

e.g., [27, 96]). There are several ways of defining what it means for a matching to be *almost stable* (see e.g., [27]). One of the better accepted notions (e.g., [33, 83, 96]) is to count the number of blocking pairs[1] - the fewer the blocking pairs, the more stable the matching. Several experimental works on parallel algorithms for the stable matching problem provide evidence that after a constant number of rounds, the number of blocking pairs can be made arbitrarily small. (e.g., [64, 88, 104]). Floréen et al. [33] showed that in the special case when the lengths of both the men's and women's preference lists are bounded by a constant, there exists a distributed version of the Gale-Shapley algorithm, which can be run for a constant number of rounds and finds an almost stable matching.

The Gale-Shapley algorithm is known to be strategy-proof for the men but not for the women (e.g., [70]). Immorlica and Mahdian [47] showed that in the setting above (and also for a more general setting), the expected number of people with more than one stable spouse is vanishingly small, and with probability $1 - o(1)$, truth-telling is a dominant strategy if the other players are truthful.

In this chapter, we focus on the setting proposed in [47] - the men's preference list is of constant length, and is chosen uniformly at random. We show that we can implement the truncated Gale-Shapley algorithm (where we simply stop the Gale-Shapley algorithm after a constant number of rounds) as an LCA, and show that this results in an almost stable matching. We then use similar techniques to show that, in a more general (non-local) setting, stopping the Gale-Shapley algorithm after a constant number of rounds gives an almost stable matching as well.

8.1 Model and Main Result

We use a graph-theoretic characterization of the *stable matching* problem (see, e.g., [32, 33]). An instance of the stable marriage problem is represented by a bipartite graph $G = (M \cup W, E)$, where M represents the set of men, and W the set of women. We make the conventional assumption that $|M| = |W| = n$. Each man has a preference list over the women that he is connected to, and each woman has a preference list over the men she is connected to. A matching $H \subseteq E$ is a set of vertex-disjoint edges. An edge e is said to be *matched* if $e \in H$. A vertex v is matched if there is some u such

[1] A *blocking pair* is a man and a woman who both prefer to be paired with each other than with their current partner.

that $e = (u, v)$ is matched. An edge $(u, v) \in E \setminus H$ is *unstable* if it holds that (1) u is unmatched or prefers v over its match in H, **and** (2) v is unmatched or prefers u over its match in H. (An unstable edge is often referred to as a *blocking pair*). A matching H is *stable* if there are no unstable edges. The stable matching problem where each man has a degree of k and his adjacent edges are chosen uniformly at random is called *k-uniform*. Note that in this case, the women's preference list lengths are binomial random variables whose value is determined by the random choices of the men, and can therefore have any length in $[n]$.

The Gale-Shapley algorithm finds a stable matching in the k-uniform setting (e.g., [35]). To ensure the locality of our algorithm, we allow our mechanism to find an almost stable matching. To do this, we allow our mechanism to "disqualify" men, in which case they remain unmatched, but are unable to contest the matching (the number of disqualified men is exactly the number of blocking pairs). We try to keep the number of disqualified men to a minimum. Our main result is the following.

Theorem 8.1.1. *Let $A = (M, W, P)$ be a stable matching problem, $|M| = |W| = n$, in the k-uniform setting. For any $\epsilon > 0$, there is a deterministic $(O(\log n), O(\log n), 0, O(\log n), 0)$-local computation mechanism for A that finds a matching with at most ϵn disqualified men and in which at most $\frac{2n}{k} + \epsilon n$ of the men remain unmatched.*

We begin by describing a non-local algorithm, AbridgedGS, and then show how to simulate it locally by a local algorithm, LocalAGS.

8.1.1 AbridgedGS

Let AbridgedGS be the Gale-Shapley men's courtship algorithm, where the algorithm is stopped after ℓ rounds, and the men rejected on that round are left unmatched. That is, in each round, each unassigned man approaches to the highest ranked woman that has not (yet) rejected him. Each woman then tentatively accepts the man she prefers out of the men who approached her, and rejects the rest. This continues until the ℓ^{th} round, and the men who were rejected on the ℓ^{th} round are left unmatched; we say that these men are *disqualified*. Note that the set of disqualified men may be a strict subset of the set of unmatched men: men who were rejected k times before the ℓ^{th} round are unmatched as well. We simulate AbridgedGS on k-uniform stable matching problems to obtain the following LCA.

8.1.2 LocalAGS - an LCA Implementation of AbridgedGS

Define the *distance* between two people to be the length of the shortest path between them in the graph. Define the d-neighborhood of a person v to be everyone at a distance at most d from v, denoted $N_d(v)$

Assume that we are queried on a specific man, m_1. We simulate AbridgedGS locally as follows: Choose some constant ℓ, whose exact value will be determined later. For each man in the 2ℓ-neighborhood of m_1, (i.e., for all m_i such that $m_i \in N_{2\ell}(m_1)$), we simulate round 1 of AbridgedGS. That is, each one approaches his preferred woman, and is either tentatively accepted or rejected . Then, for each man $m_i \in N_{2\ell-2}(m_1)$, we simulate round 2. And so on, until for $m_i \in N_2(m_1)$, (that is, m_1 and his closest male neighbors), we simulate round ℓ. We return the woman to whom m_1 is paired, "unassigned" if he was rejected by k women, and "disqualified" if he was rejected by a woman in round ℓ. We denote this algorithm LocalAGS.

In order to prove Theorem 8.1.1, we need to prove several things: that LocalAGS correctly simulates AbridgedGS and that its running time and space are bounded by $O(\log n)$ (Subsection 8.1.3), and that "not too many" men are left unmatched or disqualified (Subsection 8.1.4).

8.1.3 Correctness and Complexity

The following claim shows that the steps executed by LocalAGS are sufficient to correctly determine the output of AbridgedGS when queried on m_1.

Claim 8.1.2. *For any two men, m_i and m_j, whose distance from each other is greater than 2ℓ, m_i's actions cannot affect m_j if Algorithm* AbridgedGS *terminates after ℓ rounds.*

Proof. The proof is by induction. For $\ell = 1$, let w_1 be m_j's first choice. Only men for whom w_1 is their first choice can affect m_j, and these are a subset of the men at distance 2 from m_j. For the inductive step, assume that the claim holds for $\ell - 1$. Assume by contradiction that there is a man m_i whose actions can affect m_j within ℓ rounds, who is at a distance of at least $2\ell + 2$ from m_j. From the inductive claim, none of m_i's actions can affect any of m_j's neighbors within $\ell - 1$ rounds. As their actions in round $\ell - 1$ (or any previous round) will not be affected by m_i, and they are the only ones who can affect m_j in round ℓ, it follows that m_i cannot affect m_j within ℓ

rounds. □

We notice that AbridgedGS is, in fact, a constant-time distributed algorithm (assuming ℓ is a constant). By Claim 2.3.5, the graph is $(n/2)$-conditionally $2d$-light, and we can apply Theorem 3.2.2 to obtain the following.

Lemma 8.1.3. *The number of probes, running time and transient space requirements of algorithm* LocalAGS *is* $O(\log n)$ *per query with probability at least* $1 - \frac{1}{n^2}$.

8.1.4 Bounding the Number of Men Removed

In this section we prove that "not too many" men remain unmatched. There are two possible reasons for a man to be unmatched by LocalAGS: (1) he had already been rejected k times by round ℓ (hence he never reaches round ℓ), or (2) he was rejected (and hence disqualified) on round ℓ. We upper the probability of both (Lemma 8.1.5 and Corollary 8.1.9, respectively), and apply a union bound, to obtain the following result.

Lemma 8.1.4. *For any* $\epsilon > 0$, *setting* $\ell = \frac{2k}{\epsilon}$ *in Algorithm* LocalAGS *ensures that at most* $\frac{2n}{k} + \epsilon n$ *men remain unmatched with probability at least* $1 - \frac{1}{n^2}$.

8.1.4.1 Removal due to short lists

We bound the number of unassigned women as a result of the fact that the lists are short, noting that the number of unassigned women equals the number of unassigned men. This is given by the following lemma.

Lemma 8.1.5. *In the* k*-uniform setting, the Gale-Shapley algorithm results in at most* $\frac{2n}{k}$ *men being unassigned, with probability at least* $1 - \frac{1}{n^2}$.

Before proving Lemma 8.1.5, we will require a few preliminaries. We use the principle of deferred decisions: instead of "deciding" on the preference lists in advance, each man chooses the $(i+1)^{th}$ woman on his list only if he is rejected by the i^{th} - this is known to be equivalent to the choices being made in advance (e.g., [52]).

Consider the following stochastic process, denoted Ξ: In each round t, the (randomized) assignment function f^t is given the matching of the previous round, H^{t-1}, and assigns each man $m \in M$ a woman $w \in W$, such that if m was matched in H^{t-1}, he is assigned the same woman (i.e., if $(m, w) \in H^{t-1}$, then $f^t(m) = w$); if he is unmatched

in H^{t-1}, he is assigned a woman uniformly at random. Let $S^t(w)$ be the set of men assigned woman w by f^t, i.e., $S^t(w) = \{m : f^t(m) = w\}$. For every woman w such that $S^t(w) \neq \emptyset$, a single $m \in S^t(w)$ is chosen arbitrarily to be w's match in H^t, i.e., $(w, m) \in H^t$. The process is intialized with $H^0 = \emptyset$, and iterates for k rounds.

Remark 8.1.6. *Note that a man can choose the same woman more than once. Compare this to the case each man can approach each woman once: If a man approaches a woman he had already approached, she must be matched, and hence in this case, the men have a lower probability of approaching an unassigned woman. Therefore the number women that are unassigned at the end of this process is an upper bound to the number of unassigned women in the system where men can only approach each woman once.*

Let X_j^t be the indicator variable that is 1 if woman j is unassigned after round t, i.e., there does not exists a man $m \in M$ such that $(m, w) \in H^t$. Let $X^t = \sum_{j=1}^{n} X_j^t$ be the number of unassigned women after round t.

Proof of Lemma 8.1.5. The time required by the stochastic process Ξ described above is at most the time required by the Gale-Shapley algorithm with short lists in the k-uniform setting (from Remark 8.1.6 and the fact that it may be stopped prematurely). Hence it suffices to prove that for any constant t,

$$\Pr[X^t > \frac{2n}{t}] \leq \frac{t}{n^3}.$$

The proof is by induction. The base of the induction, $t = 1$, is immediate. For the inductive step, assume that after round t, $X^t = n/\mu$ (for some $\mu > 0$). In round $t + 1$, $\mathbb{E}[X^{t+1} | X^t = \frac{n}{\mu}] = \frac{n}{\mu}(1-1/n)^{n/\mu}$, because each unassigned man approaches any woman with probability $1/n$; hence, the probability that a specific woman is not approached by any man is $(1 - 1/n)^{n/\mu}$.

For the rest of the proof, assume that $X^t \leq \frac{2n}{t}$, and fix X^t to be some such value. We get

$$\mathbb{E}[X^{t+1} | X^t \leq \frac{2n}{t}] \leq \frac{2n}{t}(1 - 1/n)^{2n/t} < \frac{n}{t/2 \cdot e^{2/t}} < \frac{2n}{t+2}, \tag{8.1}$$

using $e^x > 1 + x$.

It remains to show that X^{t+1} is concentrated around its mean. To do so, we will define a specific martingale and use Azuma's inequality (Lemma A.1.2).

Order the men arbitrarily, $M = \{1, 2, \dots n\}$. Let M_i be the set of the first i men in the ordering: $M_i = \{1, 2, \dots i\}$. Fix some matching H^t. For some realization g of the assignment function f^{t+1}, define the following martingale

$$Y_i^{t+1}(H^t, g) = \mathbb{E}[X^{t+1}|H^t, f^{t+1}(j) = g(j) \text{ for all } j \in M_i],$$

In other words, $Y_i^{t+1}(H^t, g)$ is the expected number of unassigned women at round $t+1$, given that the matching at round t was H^t, where the expectation is taken over all realizations of f^{t+1} that agree with g on the first i men. Note that $Y_0^{t+1}(H^t, g)$ is the expected value of X^{t+1} over all possible realizations of f^{t+1}; that is, the expected number of unmatched women after $t+1$ rounds. $Y_n^{t+1}(H^t, g)$ is simply the number of unmatched women after $t+1$ rounds when the allocation function is g. X^{t+1} satisfies the Lipschitz condition, because if two realizations of f^{t+1}, say f' and f'', only differ on the allocation of a single man, $\left|X^{t+1}|f' - X^{t+1}|f''\right| \leq 1$ (where $X^{t+1}|f$ denotes the realization of X^{t+1} given that f is the realization of f^{t+1}). Therefore, (see [5]),

$$|Y_{i+1}^{t+1}(H^t, g) - Y_i^{t+1}(H^t, g)| \leq 1.$$

We can therefore apply Lemma A.1.2 (Azuma's inequality):

$$\Pr[|X^{t+1} - \mathbb{E}[X^{t+1}]| > \lambda\sqrt{n}] < 2e^{-\lambda^2/2}.$$

Setting $\lambda = \frac{2\sqrt{n}}{(t+1)(t+2)}$, we have that

$$\Pr\left[|X^{t+1} - \mathbb{E}[X^{t+1}]| > \frac{2n}{(t+1)(t+2)}\right] < 2e^{-2n/t^4},$$

for $t \geq 2$.

Therefore, since we assume that $X^t \leq \frac{2n}{t}$ and hence, by Equation (8.1), $\mathbb{E}[X^{t+1}] < \frac{2n}{t+2}$, it holds that

$$\Pr\left[X^{t+1} > \frac{2n}{t+1}|X^t \leq \frac{2n}{t}\right] < 2e^{-2n/t^4} < \frac{1}{n^3}, \tag{8.2}$$

for $t \leq \left(\frac{6n}{\log n}\right)^{1/4}$. By the inductive hypothesis

$$\Pr\left[X^t > \frac{2n}{t}\right] \leq \frac{t}{n^3}. \tag{8.3}$$

Therefore, using Equations (8.2), and (8.3), we have

$$\begin{aligned}
\Pr\left[X^{t+1} > \frac{2n}{t+1}\right] =& \Pr\left[X^{t+1} > \frac{2n}{t+1} | X^t \leq \frac{2n}{t}\right] \Pr\left[X^t \leq \frac{2n}{t}\right] \\
&+ \Pr\left[X^{t+1} > \frac{2n}{t+1} | X^t > \frac{2n}{t}\right] \Pr\left[X^t > \frac{2n}{t}\right]. \\
\leq& \frac{t}{n^3} + \frac{1}{n^3} \\
=& \frac{t+1}{n^3}.
\end{aligned}$$

□

8.1.4.2 Removal due to the number of rounds being limited

Because we stop the LocalAGS algorithm after a constant (ℓ) number of rounds, it is possible that some men who "should have been" matched are disqualified because they were rejected by their i^{th} choice in round ℓ ($i < k$). We show that this number cannot be very large.

Let R_r denote the number of men rejected in round $r \geq 1$.

Observation 8.1.7. *R_r is monotone decreasing in r.*

Lemma 8.1.8. *The number of men rejected in round r is at most $\frac{nk}{r}$.*

Proof. As each man can be rejected at most k times, the total number of rejections possible is kn. The number of men who can be rejected in round r is at most

$$\begin{aligned}
R_r \leq& kn - \sum_{j=1}^{r-1} R_j \\
\Rightarrow R_r \leq& kn - (r-1)R_r \qquad (8.4) \\
\Rightarrow R_r \leq& n\frac{k}{r}.
\end{aligned}$$

Where Inequality (8.4) is due to monotonicity of R_r, i.e., Observation 8.1.7. □

Corollary 8.1.9. *For any $\epsilon > 0$, setting $r = \frac{k}{\epsilon}$ ensures that the number of men rejected in round r is at most ϵn.*

8.2 Some General Properties of the Gale-Shapley Algorithm

We use the results and ideas of Section 8.1 to prove some interesting features of the (general) Gale-Shapley stable matching algorithm, when the mens' lists are of length at most k. (These results immediately extend to our local version of the algorithm, LocalAGS.) Note that the proof of Lemma 8.1.8 makes no assumption on how the men's selection is made, and therefore, Lemma 8.1.8 implies that as long as each man's list is bounded by k, if we run the Gale-Shapley for ℓ rounds, at most $\frac{nk}{\ell}$ men will be rejected in that round. This immediately gives us an additive approximation bound for the algorithm if we stop after ℓ rounds:

Corollary 8.2.1 (to Lemma 8.1.8)**.** *Assume that the output of the Gale-Shapley algorithm on a stable matching problem, where the preference lists of the men are of length at most k, is a matching of size M^*. Then, stopping the Gale Shapley algorithm after ℓ rounds will result in a matching of size at least $M^* - \frac{nk}{\ell}$.*

We would like to also provide a multiplicative bound. Again, we assume that the mens' list length is bounded by k, but make no other assumptions. For each round i, let M_i be the size of the current matching; let D_i be the number of men who have already approached all k women on their list and have been rejected by all of them; let C_i be the number of men who were rejected by women in round i, but have approached fewer than k women so far; as before, let R_i be the number of men rejected in round i. Denote the size of the matching returned by the Gale-Shapley algorithm (if it were to run to completion) by M^*.

Claim 8.2.2. $C_{k+1} \leq kM^*$.

Proof. Note that $R_i = C_i + D_i - D_{i-1}$. For $i < k, D_i = 0$. As M_i is monotonically increasing in i, $\forall i \leq k, R_i \geq n - M^*$.

$$\sum_{i=1}^{k} R_i \geq kn - kM^*.$$

Hence,

$$C_{k+1} \leq kn - \sum_{i=1}^{k} R_i \leq kM^*.$$

□

Corollary 8.2.3. *For every $\epsilon > 0$, there exists a constant $\ell > 0$ such that $C_\ell \leq \epsilon M^*$.*

Proof. Denote the maximum number of total rejections possible from round i onwards by L_i. Clearly,

$$L_i \leq k(M_i + C_i) \leq k(M^* + C_i).$$

For all i such that $C_i \geq \epsilon M^*$, we have

$$L_i \leq \left(1 + \frac{1}{\epsilon}\right) kC_i.$$

Therefore, from Claim 8.2.2,

$$L_{k+1} \leq \left(1 + \frac{1}{\epsilon}\right) k^2 M^*.$$

Putting everything together, we have,

$$\begin{aligned}
L_{i+1} &\leq L_i - C_i \\
\Rightarrow L_{i+1} &\leq L_i \left(1 - \frac{1}{k(1+\frac{1}{\epsilon})}\right) \\
\Rightarrow L_{k+i+1} &\leq L_{k+1} \left(1 - \frac{1}{k(1+\frac{1}{\epsilon})}\right)^i \\
&\leq \left(1 + \frac{1}{\epsilon}\right) k^2 M^* \left(1 - \frac{1}{k(1+\frac{1}{\epsilon})}\right)^i \\
&\leq 2k^2 M^* e^{-\frac{i}{k(1+\frac{1}{\epsilon})}}.
\end{aligned}$$

Taking $i = k(1 + \frac{1}{\epsilon}) \log \frac{2k^2}{\epsilon}$ gives $C_{k+i+1} \leq L_{k+i+1} \leq \epsilon M^*$. □

This gives us,

Theorem 8.2.4. *Consider a stable matching problem. Let the length of each man's list be bounded by k. Denote the size of the stable matching returned by the Gale-Shapley algorithm by M^*. Then, if the process is stopped after $O(\frac{k}{\epsilon} \log \frac{k}{\epsilon})$ rounds, the matching*

returned is at most a $(1+\epsilon)$*-approximation to* M^**, and has at most* ϵM^* *unstable couples.*

As a corollary to Theorem 8.2.4, when the men's and women's list lengths are both bounded by a constant, there is an LCA that runs in constant time an provides a matching with at most ϵ unstable edges that is a $(1+\epsilon)$ approximation to the matching returned by the Gale-Shapley algorithm.

Corollary 8.2.5. *If both men and women have lists of length at most* k*, then for any* ϵ *there is an* $(O(1), O(1), 0, O(1), 0)$*-LCA for stable matching that returns a matching that is at most a* $(1+\epsilon)$*-approximation to the matching returned by the Gale-Shapley algorithm, and with at most an* ϵ*-fraction of the edges being unstable.*

Chapter 9

Machine Scheduling

Consider the following job scheduling problem: n identical jobs arrive online and need to be allocated to m identical machines, with the objective of minimizing the makespan - the maximal load on any machine. Azar et al. [9] proposed the following algorithm: each job chooses, uniformly at random, d machines, and allocates itself to the least loaded machine from its d choices. They showed that the maximal load is $\Theta(n/m) + (1 + o(1)) \ln \ln m / \ln d$. A large volume of work has been devoted to variations on this problem, such as having weighted jobs [102]; and variations on the algorithm, such as the non-uniform job placement strategies of [108]. Of particular relevance to this work is the case of non-uniform machines: Berenbrink et al. [14] showed that in this case the maximum load can also be bounded by $\Theta(n/m) + O(\ln \ln m)$.

The classical off-line job scheduling problem has two main variations: (1) Related machines, where each job i takes a certain amount time, t_i, to complete, regardless of which machine it is allocated, and (2) Unrelated machines, where each job i takes time $t_{i,j}$ to complete on machine j. Both problems are known to be NP-hard. Hochbaum and Shmoys [44] showed a PTAS for scheduling on related machines. Lenstra et al. [58], presented a 2-approximation algorithm for scheduling on unrelated machines and showed that the optimal allocation is not approximable to within $\frac{3}{2} - \epsilon$ (unless $P = NP$). The problem of finding a truthful mechanism for scheduling (on unrelated machines) was introduced by Nisan and Ronen [81], who showed an m-approximation to the problem, and a lower bound of 2. Archer and Tardos [8] were the first to tackle the related machine case; they showed a randomized 3-approximation polynomial algorithm and a polynomial pricing scheme to derive a mechanism that is truthful in expectation. Since then, much work has gone into finding mechanisms with improved approximation

ratios, until Christodoulou and Kovács [20] settled the problem by showing a deterministic PTAS, and a corresponding mechanism that is deterministically truthful.

In this chapter, we consider related machines. We show two local mechanisms: one for the more general setting that is truthful in expectation, and one for the restricted case (i.e., when each job can run on one of at most a constant number of predetermined machines) that is universally truthful. Both mechanisms provide an $O(\log \log n)$-approximation to the optimal makespan.

9.1 The Model

We consider the following (off-line) job scheduling setting. There is a set $\mathcal{I}$ of m machines (or "bins") and a set $\mathcal{J}$ of n uniform jobs (or "balls"). Each machine $i \in \mathcal{I}$ has an associated capacity c_i (sometimes referred to as its "speed"). We assume that the capacities are positive integers. Given that h_i jobs are allocated to machine i, its *load* is $\ell_i = h_i/c_i$. (h_i is also called the *height* of machine i.) The *utility* of machine i is quasi-linear, namely, when it has load ℓ_i and receives payment p_i then its utility is $u_i(\ell_i, p_i) = p_i - \ell_i$.

The *makespan* of an allocation is $\max_i\{\ell_i\} = \max_i\{h_i/c_i\}$. In our setting, the players are the machines and their private information is their true capacities. Each machine i submits a *bid* b_i (which represents its capacity). The mechanism designer would like to elicit from the machines the true information about their capacities in order to be able to minimize the makespan of the resulting allocation. We assume that the capacities of the machines cannot depend on the number of machines or jobs in the system (i.e., that the bids of the machines are independent of m or n), and hence are upper bounded by some constant. Although we feel this is a reasonable assumption, in Remark 9.2.6, we show that in some cases we can relax it.

For any allocation algorithm $\mathcal{A}$, and bid vector b, define $\mathcal{A}(b) = (\mathcal{A}^1(b), \ldots, \mathcal{A}^j(b), \ldots, \mathcal{A}^n(b))$ to be the allocation vector, which, when given b as an input, assigns each job j to a machine $i = \mathcal{A}^j(b)$. When the bids b_{-i} (the bids of all machines except for i) are fixed, we sometimes omit them from the notation for clarity.

Definition 9.1.1. *(Monotonicity) A randomized allocation function $\mathcal{A}$ is* monotone in expectation *if for any machine i, and any bids b_{-i}, the expected load of machine i, $\mathbb{E}[\ell_i(b_i, b_{-i})]$, is a non-decreasing function of b_i.*

A randomized allocation function $\mathcal{A}$ is universally monotone *if for any machine i, and any bids b_{-i}, the load of machine i, $\ell_i(b_i, b_{-i})$, is a non-decreasing function of b_i for any realization of the randomization of the allocation function.*

Given an allocation function $\mathcal{A}$, we would like to provide a payment scheme $\mathcal{P}$ to ensure that our mechanism $\mathcal{M} = (\mathcal{A}, \mathcal{P})$ is truthful. It is known that a necessary and sufficient condition is that the allocation function $\mathcal{A}$ is monotone ([74]; see also [8]).

Theorem 9.1.2. *[74] The allocation algorithm $\mathcal{A}$ admits a payment scheme $\mathcal{P}$ such that the mechanism $\mathcal{M} = (\mathcal{A}, \mathcal{P})$ is truthful-in-expectation (universally truthful) if and only if $\mathcal{A}$ is monotone in expectation (universally monotone).*

We consider two load balancing settings: The *standard* setting (cf. [14, 110]) is a slight variation on the basic power-of-d choices setting proposed in [9]. Let $d \geq 2$ be some integer. For each job j, the mechanism chooses a subset $I_j \subseteq \mathcal{I}$, $|I_j| = d$ of machines that the job can be allocated to. The probability that machine $i \in I_j$ is proportional to b_i (specifically, it is $\frac{db_i}{\sum_i b_i}$). In the *restricted* setting (cf. [10]), each job can be allocated to a subset of at most d machines, where the subsets I_j are given as an input to the allocation algorithm. The restricted setting models the case when the jobs have different requirements, and there is only a small subset of machines that can run each job. We restrict our attention to the case when the number jobs $n = \Theta(C)$ jobs where C is the total capacity of the machines. This is a standard assumption, as it is considered to be the worst case scenario (see e.g., [9, 14, 110]), and so a solution for this case implies that there is an equally good solution for all other cases as well.

In both of these settings our results rely on a reduction to an on-line algorithm for the problem. This is summarized in the the following theorem, which is an adaptation of Theorem 4.5.2.

Theorem 9.1.3. *Consider a job scheduling problem for n jobs and $\Theta(n)$ machines. For each job j, there is a constant-size subset of machines I_j, chosen uniformly at random, and j cannot be allocated to any machine $i \notin I_j$. For any crisp on-line algorithm $\mathcal{LB}$, there exists an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA that, when queried on a job, allocates it to a machine, such that the resulting allocation is consistent with that of $\mathcal{LB}$.*

Using known crisp online load balancing algorithms, we can get LCAs for many problems, such as the following.

Corollary 9.1.4. *(Using [13]) Suppose we wish to allocate m balls into n bins of uniform capacity, $m \geq n$, where each ball chooses d bins independently and uniformly at random. There exists a $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA that allocates the balls in such a way that the load of the most loaded bin is $m/n + O(\log \log n / \log d)$ w.h.p.*

Corollary 9.1.5. *(Using [108]) Suppose we wish to allocate n balls into n bins of uniform capacity, where each ball chooses d bins independently at random, one from each of d groups of almost equal size $\theta(\frac{n}{d})$. There exists a $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA that allocates the balls in such a way that the load of the most loaded bin is $\ln \ln n/(d-1)\ln 2 + O(1)$ w.h.p.* [1]

Corollary 9.1.6. *(Using [14]) Suppose we wish to allocate m balls into $n \leq m$ bins, where each bin i has a capacity c_i, and $\sum_i c_i = m$. Each ball chooses d bins at random with probability proportional to their capacities. There exists a $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA that allocates the balls in such a way that the load of the most loaded bin is $2\log \log n + O(1)$ w.h.p.*

Corollary 9.1.7. *(Using [17]) Suppose we have n bins, each represented by one point on a circle, and n balls are to be allocated to the bins. Assume each ball needs to choose $d \geq 2$ points on the circle, and is associated with the bins closest to these points. There exists a $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA that allocates the balls in such a way that the load of the most loaded bin is $\ln \ln n / \ln d + O(1)$ w.h.p.*

9.2 A Mechanism for the Standard Setting

In the standard setting, each machine i has an integer capacity c_i. One way of modeling this is to think of the allocation field as consisting of $\sum_i c_i$ *slots* of size 1, where machine i "owns" c_i slots. Recall that a machine's *height* is the number of jobs that are allocated to it; the *load* of machine i is its height divided by c_i. The *virtual load* of machine i is its height divided by its bid b_i. Given the bids b of the machines, let $B = \sum_{i=1}^{n} b_i$. An allocation algorithm allocates jobs to slots: when a job j is allocated to a specific slot, the machine that owns the slot receives j. We provide the following simple on-line allocation algorithm $\mathcal{A}_{SLMS}$, which is modeled on the algorithm presented in [14].

[1] In fact, in this setting the tighter bound is $\frac{\ln \ln n}{d \ln \phi_d} + O(1)$, where ϕ_d is the ratio of the d-step Fibonacci sequence, i.e. $\phi_d = \lim_{k\to\infty} \sqrt[k]{F_d(k)}$, where for $k < 0$, $F_d(k) = 0$, $F_d(1) = 1$, and for $k \geq 1$ $F_d(k) = \sum_{i=1}^{d} F_d(k-i)$

1. Choose for job j a subset I_j of d slots out of B, where each slot has equal probability. (I_j may include different slots owned by the same machine.)

2. Given I_j, job j is allocated to the lowest slot (i.e., the one containing the fewest jobs) in I_j (breaking ties uniformly at random). Slots are treated as being independent of their machines. That is, it is possible that if a job chooses two slots a and b, which belong to machines A and B, a has fewer jobs than b, but B has a higher (virtual) load than A.

Note: Although it may not be possible to compute B locally exactly, it has been shown in that an approximate calculation suffices (e.g., [17, 110]); therefore, for simplicity, we assume that it is possible to compute B locally.

Lemma 9.2.1. *The randomized allocation algorithm $\mathcal{A}_{SLMS}$ is monotone in expectation.*

Proof. Let $B = \sum_j b_j$ and $B_{-i} = \sum_{j \neq i} b_j$. Since all the slots are identical, by symmetry the expected number of jobs allocated to each slot is exactly n/B. The expected height of machine i is therefore

$$\mathbb{E}[h_i(b_i)] = \frac{b_i}{B_{-i} + b_i} n,$$

which is monotone increasing in b_i (for $b_i, B_{-i} \geq 0$). □

From Theorem 9.1.2, we conclude:

Lemma 9.2.2. *The randomized allocation function $\mathcal{A}_{SLMS}$ admits a payment scheme $\mathcal{P}_{SLMS}$ such that the mechanism $\mathcal{M}_{SLMS} = (\mathcal{A}_{SLMS}, \mathcal{P}_{SLMS})$ is truthful in expectation.*

It is interesting to note that the above algorithm does not admit a universally truthful mechanism. To show this, we prove a slightly stronger claim, which we then adapt to our setting: the GREEDY algorithm, in which each job chooses d machines at random, and is allocated to the least loaded among them (post-placement)[2], breaking ties arbitrarily), does not admit a universally truthful mechanism.

Claim 9.2.3. *Algorithm* GREEDY *is not universally monotone.*

[2]That is, the load is computed including the allocation of the arriving job. For example, if machine A has capacity 4 and height 2 and machine B has capacity 16 and height 9, the job will go to machine B, as after placing the job, the load on B would 10/16, compared to 3/4 on A.

Proof. Assume we have 4 machines: A, B, C, and D, with bids 4, 4, 8 and 1 respectively. The first 2 jobs choose machines A and D (which we abbreviate to AD), the next 2 jobs choose BD, and the next 6 jobs choose CD. After these 10 jobs, the heights of the machines are $(2, 2, 6, 0)$ (recall that the Greedy algorithm allocates according to the *post-placement* load). The 11^{th} job chooses AB, and the 12^{th} job chooses AC. As ties are broken at random, assume machine A receives job 11. Machine C then receives job 12, making the capacities $(3, 2, 7, 0)$.

Now assume machine C bids 9, and the choices of the first 10 jobs and the 12^{th} job remain the same, but because C bid higher, now the 11^{th} job chooses C instead of A (so job 11 chooses BC). Now machine B receives the 11^{th} job and machine A receives the 12^{th} job, making the capacities $(3, 3, 6, 0)$. Machine C received less jobs although it bid more! □

It is easy to adapt the above proof to Algorithm $\mathcal{A}_{SLMS}$: instead of choosing machines, each job chooses 2 slots. So the first job will choose slot 1 of machine A and the only slot of machine D; the second job will choose slot 2 of machine A and machine D's slot; and so on. This gives the following corollary.

Corollary 9.2.4. *Algorithm $\mathcal{A}_{SLMS}$ is not universally monotone.*

By Theorem 9.1.3, the allocation function $\mathcal{A}_{SLMS}$ can be transformed to an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA . For clarity, we overload the notation, letting $\mathcal{A}_{SLMS}$ represent both the on-line allocation algorithm and its respective LCA, as it is easy to distinguish between them from context. We would now like to show a payment scheme $\mathcal{P}_{SLMS}$ such that the mechanism $\mathcal{M}_{SLMS} = (\mathcal{A}_{SLMS}, \mathcal{P}_{SLMS})$ is a local mechanism. We need to show a payment scheme that can be implemented as an LCA and guarantees truthfulness. We give a deterministic payment scheme, that is similar to the payments schemes of [7] and [14]. We also comment on the possibility of a randomized payment scheme when the bids can depend on the total capacity. The randomized payment scheme is similar to that of [11].

Lemma 9.2.5. *If the bids of the machines are bounded by a constant, there exists a deterministic local payment scheme $\mathcal{P}_{SLMS}$ such that the mechanism $\mathcal{M}_{SLMS} = (\mathcal{A}_{SLMS}, \mathcal{P}_{SLMS})$ is truthful in expectation.*

Proof. Archer and Tardos [8] showed that the following payment scheme makes for a

truthful mechanism fulfilling voluntary participation. For bid b_i:

$$p_i(b_i, b_{-i}) = b_i h_i(b_i, b_{-i}) + \sum_{x=0}^{b_i} h_i(x, b_{-i})dx \ . \tag{9.1}$$

As b_i is bounded by a constant, we can execute $\mathcal{A}_{SLMS}$ with all values of $b_i \in [0, b_i]$, to compute p_i. This takes a constant number of executions of $\mathcal{A}_{SLMS}$. □

Remark 9.2.6. *If b_i is not necessarily a constant, but the mechanism has access to the value B_{-i}, there is a randomized payment scheme that we can use. Equation (9.1) is the expected payment. From symmetry, $\mathbb{E}(h_i(B)) = \frac{b_i}{B}$, hence we can rewrite Equation (9.1) as*

$$p_i(b_i, b_{-i}) = n\frac{b_i^2}{B_{-i} + b_i} + n\sum_{x=0}^{b_i} \frac{x}{B_{-i} + x}.$$

Choose, uniformly at random, $k \in [1, b_i]$, and take the payment to be

$$n\frac{b_i^2}{B} + nb_i \cdot \frac{k}{B_{-i} + k}$$

This gives the correct expected payment, and takes $O(1)$ time.

Berenbrink et al. [14], showed that $\mathcal{A}_{SLMS}$ provides an $O(\log \log m)$ approximation to the optimal makespan. Therefore, by Theorem 9.1.3, the LCA of $\mathcal{A}_{SLMS}$ provides the same approximation ratio. Combining Lemma 9.2.2, and Lemma 9.2.5, we state our main result for the standard setting:

Theorem 9.2.7. *There exists an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-local mechanism to scheduling on related machines in the standard setting that is truthful in expectation, and provides an $O(\log \log n)$-approximation to the makespan.*

9.3 A Mechanism for the Restricted Setting

In the restricted setting, each job can only be allocated to one of a set $I_j \subseteq \mathcal{I}$ of d machines (i.e., $|I_j| = d$). As opposed to the standard setting, I_j is not selected by the mechanism, but is part of the input. We assume that these sets are selected i.i.d. from all possible sets, and the probability of machine i to be in I_j is proportional to its capacity c_i. The first assumption is necessary for bounding the running time, the

second to guarantee the approximation ratio. The second requirement can be relaxed slightly, (see e.g. [110]) but for clarity of the proofs, we will assume that it holds exactly. Similarly to the previous subsection, we assume that the capacity of each machine is bounded by a constant.

We define the (on-line) algorithm $\mathcal{A}_{RLMS}$ for assigning jobs to machines as follows. Initially, a permutation π of the machines is selected arbitrarily, for tie-breaking. Job t is assigned to the machine $i \in I_j$ for which the *post-placement load*, $lp_i^{t+1}(b_i) = \lfloor \frac{h_i^t(b_i)+1}{b_i} \rfloor$ is smallest, breaking ties according to π. The following claim shows why it is necessary to take the floor of the load (in other words, why we cannot define $lp_i^{t+1}(b_i) = \frac{h_i^t(b_i)+1}{b_i}$): the Greedy algorithm with $lp_i^{t+1}(b_i)$ defined this way does not admit a universally truthful mechanism.

Claim 9.3.1. *The (unmodified)* Greedy *algorithm is not universally monotone in the restricted case.*

Proof. Assume we have 3 machines A, B, C, with bids $(4, 8, 36)$ respectively, and a tie-breaking permutation: $A < B < C$ (jobs always prefer machine A to machines B and C, and machine B to machine C). The allocation at time t is $(1, 3, 18)$. The next job's restricted set contains machines A and B (which we abbreviate to AB), and the following two jobs' sets are BC and AB respectively. The first job is allocated to A (since the post-placement loads on A and B are $2/4$ and $4/8$ respectively, hence we use the tie-breaking rule). The second job is allocated to B ($4/8 < 19/36$) and the third job to B ($5/8 > 3/4$). The heights of the machines are now $(2, 5, 18)$.

Now assume B declares its capacity to be 9, and assume that at time t, there is no difference in the allocation (it is easy to verify that this is indeed possible). The loads at time t in this case are: $1/4, 3/9, 18/36$. The jobs' choices are part of the input to the mechanism, so are unaffected by the bids, and remain AB, BC, AB. The first job is allocated to B ($2/4 > 4/9$), the second job to C ($19/36 < 20/36 = 5/9$), and the third job to A ($2/4 < 5/9$). The heights of the machines are now $(2, 4, 19)$. Thus, B receives less jobs despite bidding higher. □

Interestingly, although Greedy is not universally monotone, $\mathcal{A}_{RLMS}$ is.

Theorem 9.3.2. *For any permutation π of the machines and any job arrival order, the allocation function $\mathcal{A}_{RLMS}$ is universally monotone increasing in the machines' bids.*

From the definition of universal monotonicity (Definition 9.1.1), it suffices to prove the following lemma:

Lemma 9.3.3. *For any machine i, fixing b_{-i}, for any $b'_i > b_i$, we have that $h_i(\mathcal{A}_{RLMS}(b'_i, b_{-i})) \geq h_i(\mathcal{A}_{RLMS}(b_i, b_{-i}))$.*

To prove Lemma 9.3.3, define $D^t(k, b'_i, b_i)$ to be the difference in the number of jobs allocated to machine k between $\mathcal{A}_{RLMS}(b'_i)$ and $\mathcal{A}_{RLMS}(b_i)$ up to and including the arrival of job t (which we call *time* t). We abbreviate this to $D^t(k)$ when b'_i and b_i are clear from the context. (If machine k received less jobs, then $D^t(k)$ is negative.) We say that machine k *steals* a job from machine l at time t if $\mathcal{A}^t_{RLMS}(b_i) = l$ and $\mathcal{A}^t_{RLMS}(b'_i) = k$. We will show that the only machine for which $D^t(k)$ can be positive at some time t is machine i, therefore, as $\sum_{j=1}^{n} D^t(j) = 0$, we have that $D^t(i)$ can never be negative.

Proposition 9.3.4. *For any machine i, fixing b_{-i}, if $b'_i > b_i$ then at all times t, for any machine $k \neq i$, $D^t(k) \leq 0$.*

Informally, Proposition 9.3.4 says that if bin i claims its capacity is larger than it actually is, no bin except for i can receive more balls. The following corollary follows immediately from Proposition 9.3.4, and implies Lemma 9.3.3.

Corollary 9.3.5. *For any machine i, fixing b_{-i}, if $b'_i > b_i$ then at all times t, $D^t(i) \geq 0$.*

Before proving Proposition 9.3.4, we first will make the following simple observation

Observation 9.3.6. *For any machine k, if $D^t(k) \leq 0$ then $lp^t_k(b'_i) \leq lp^t_k(b_i)$.*

Proof. For $k \neq i$, as k's bid is the same in both allocations, if it received less jobs by time t in $\mathcal{A}_{RLMS}(b_i)$ then the observation follows. If $k = i$, the observation follows since $b'_i > b_i$. □

We now prove Proposition 9.3.4:

Proof of Proposition 9.3.4. The proof is by induction on t. At time $t = 0$, $D^0(k) = 0$ for every k.

Assume the proposition is true for times $t = 0, 1, \ldots, \tau - 1$. We show it holds for $t = \tau$, by contradiction. Assume that we have a machine $k \neq i$ such that $D^\tau(k) > 0$. At time $\tau - 1$, for all $k \neq i$, by the induction hypothesis, it holds that $D^{\tau-1}(k) \leq 0$.

The only way that $D^\tau(k) > 0$ is if machine k has $D^{\tau-1}(k) = 0$ and at time τ steals a job. Assume first that machine k steals a job from machine $l \neq i$. This means that in $\mathcal{A}_{RLMS}(b_i)$, machine l received job τ, therefore

$$lp_l^\tau(b_i) \leq lp_k^\tau(b_i). \tag{9.2}$$

By Observation 9.3.6, $lp_l^\tau(b'_i) \leq lp_l^\tau(b_i)$, and so

$$lp_l^\tau(b'_i) \leq lp_l^\tau(b_i) \leq lp_k^\tau(b_i) = lp_k^\tau(b'_i).$$

If machine k steals job τ from machine l, then $lp_k^\tau(b'_i) \leq lp_l^\tau(b'_i)$. This is a contradiction to Equation (9.2) because there cannot be an equality both here and in Equation (9.2), as the tie-breaking permutation π is fixed. More precisely, if $lp_l^\tau(b'_i) = lp_l^\tau(b_i) = lp_k^\tau(b_i) = lp_k^\tau(b'_i)$, then job τ will be allocated to the same machine in b_i and b'_i, according to the permutation π.

Therefore, machine k must steal job τ from machine i, which gives us

$$lp_i^\tau(b_i) \leq lp_k^\tau(b_i) = lp_k^\tau(b'_i) \leq lp_i^\tau(b'_i). \tag{9.3}$$

The first inequality is due to the fact that machine i receives job τ in $\mathcal{A}_{RLMS}(b_i)$. The equality is due to the fact that $D^{\tau-1}(k) = 0$, and the second inequality is because machine k receives job τ in $\mathcal{A}_{RLMS}(b'_i)$. And so,

$$lp_i^\tau(b_i) < lp_i^\tau(b'_i), \tag{9.4}$$

because one of the inequalities in Equation (9.3) must be strict, as the tie-breaking permutation π is fixed.

Assume that the last time before τ that machine i stole a job is time ρ, and label by z the machine that i stole from at that time. We have

$$lp_i^\rho(b'_i) \leq lp_z^\rho(b'_i) \leq lp_z^\rho(b_i) \leq lp_i^\rho(b_i).$$

The first inequality is because machine i received job ρ in $\mathcal{A}_{RLMS}(b'_i)$. The middle inequality is because $D^\rho(z) \leq 0$. The last inequality is because machine z received job

ρ in $\mathcal{A}_{RLMS}(b_i)$. Again, at least one inequality must be strict, giving

$$lp_i^{\rho}(b_i') < lp_i^{\rho}(b_i),$$

which implies, for all $\alpha \geq 0$,

$$\left\lfloor \frac{h_i^{\rho}(b_i') + \alpha + 1}{b_i'} \right\rfloor \leq \left\lfloor \frac{h_i^{\rho}(b_i) + \alpha}{b_i} \right\rfloor, \tag{9.5}$$

since $b_i' > b_i \geq 1$.

Because job ρ was the last job that machine i stole, it received at least as many jobs between ρ and τ in $\mathcal{A}_{RLMS}(b_i)$ as in $\mathcal{A}_{RLMS}(b_i')$. Label the number of jobs i received between ρ and τ (including ρ but excluding τ) in $\mathcal{A}_{RLMS}(b_i)$ by β and in $\mathcal{A}_{RLMS}(b_i')$ by β^*.

Observation 9.3.7. $\beta^* \leq \beta + 1$.

Proof. Machine i received at least as many jobs in $\mathcal{A}_{RLMS}(b_i)$ as in $\mathcal{A}_{RLMS}(b_i')$ after ρ. This must be true because ρ was the last time machine i stole a job. However, machine i received the job at time ρ in $\mathcal{A}_{RLMS}(b_i')$ but not in $\mathcal{A}_{RLMS}(b_i)$, and so we cannot claim that $\beta^* \leq \beta$, but only that $\beta^* \leq \beta + 1$. □

Proof of Proposition 9.3.4 continued. From the definition of lp and equation (9.5), we get:

$$\begin{aligned}
lp_i^{\tau}(b_i') &= \left\lfloor \frac{h_i^{\tau}(b_i') + 1}{b_i'} \right\rfloor \\
&= \left\lfloor \frac{h_i^{\rho}(b_i') + \beta^* + 1}{b_i'} \right\rfloor && (9.6) \\
&\leq \left\lfloor \frac{h_i^{\rho}(b_i') + \beta + 2}{b_i'} \right\rfloor && (9.7) \\
&\leq \left\lfloor \frac{h_i^{\rho}(b_i) + \beta + 1}{b_i} \right\rfloor && (9.8) \\
&= lp_i^{\tau}(b_i). && (9.9)
\end{aligned}$$

Equality (9.6) stems from the definition of β^*, Inequality (9.7) is due to Observation 9.3.7, Inequality (9.8) is due to Equation (9.5), and Equality (9.9) is from the definition of β.

This is in contradiction to Equation (9.4), and therefore $D^\tau(k) \leq 0$. This concludes the proof of the proposition. □ □

Given that $\mathcal{A}_{RLMS}$ is universally monotone, we can once again use the payment scheme of Archer and Tardos [8] to obtain the following lemma.

Lemma 9.3.8. *There exists a local payment scheme $\mathcal{P}_{RLMS}$ such that the mechanism $\mathcal{P}_{RLMS} = (\mathcal{A}_{RLMS}, \mathcal{P}_{RLMS})$ is universally truthful.*

It remains to bound the approximation ratio of our algorithm.

Lemma 9.3.9. *The allocation algorithm $\mathcal{A}_{RLMS}$ provides an $O(\log\log n)$-approximation to the optimal allocation.*

The proof is similar to the proof for the unmodified Greedy algorithm in the case of non-uniform bins of [14]. We provide it for completeness. We prove the theorem for the case $d = 2$ (each job can be assigned to one of 2 machines). The proof is easily extendable to $d > 2$. For the proof (not the algorithm), we regard each machine i of capacity c_i as having c_i slots of capacity 1. Before presenting the proof we need several definitions:

The *load vector* of an allocation of jobs to m machines is $L = (\ell_1, \ldots, \ell_m)$, where ℓ_i is the load of machine i. The *normalized load vector* $\bar{L}$ consists of the members of L in non-increasing order (ties are broken arbitrarily). For the case of non-uniform machines of capacities $c_1, \ldots c_m$, and total capacity $C = \sum_{i=1}^{m} c_i$, we define the *slot-load vector* $S = (h_{1,1}, \ldots h_{1,c_1}, h_{2,1}, \ldots h_{2,c_2}, \ldots h_{n,1}, \ldots h_{n,c_n})$, where if machine i is allocated r jobs, the first r mod c slots will have $\lceil r/c \rceil$ jobs, and the remaining slots will have $\lfloor r/c \rfloor$ jobs. If a machine has an uneven allocation of jobs, we call the slots with more jobs *heavy*, and the slots with less jobs *light*. If all of the slots of the machine have an identical number of jobs assigned to them, we call all the slots *light*. When we allocate a job to a machine, we add it to one of the light slots, arbitrarily. The *normalized slot load vector* $\bar{S}$ is S sorted in non-increasing order (slots of the same machine may be separated in $\bar{S}$). We add a subscript t to these vectors, i.e., L_t, $\bar{L}_t$, S_t and $\bar{S}_t$ to indicate the vector after the allocation of the t-th job.

Definition 9.3.10 (Majorization, $\succeq$)**.** *We say that a vector $P = (p_1, \ldots, p_a)$* majorizes

vector $Q = (q_1, \ldots, q_b)$ *(denoted* $P \succeq Q$*) if and only if for all* $1 \leq k \leq \min(a, b)$*,*

$$\sum_{i=1}^{k} \bar{p}_i \geq \sum_{i=1}^{k} \bar{q}_i,$$

where $\bar{p}_i$ *and* $\bar{q}_i$ *are the* i*-th entries of the normalized vectors* $\bar{P}$ *and* $\bar{Q}$*.*

For $n \in \mathbb{N}$, let $[n]$ denote $\{1, \ldots, n\}$.

Definition 9.3.11 (System Majorization)**.** *Let* A *and* B *be two processes allocating* n *jobs to machines with total capacity* C*. Let* $\tau = (\tau_1 \ldots \tau_{2n})$*,* $\tau_i \in [C]$ *be a vector representing the (slot) choices of the* n *jobs (*τ_{2i-1} *and* τ_{2i} *are the choices of the* i*-th job). Let* $S^A(\tau)$ *and* $S^B(\tau)$ *be the slot load vectors using* A *and* B *respectively with the random choices specified by* τ*. Then we say*

1. A majorizes B *(denoted by the overloaded notation* $A \succeq B$*) if there is a bijection* $f : [C]^{2n} \to [C]^{2n}$ *such that for all possible random choices* $\tau \in [C]^{2n}$*,we have*

$$L^A(\tau) \succeq L^B(f(\tau)).$$

2. *The maximum load of* A majorizes *the maximum load of* B *(denoted by* $A \succeq_n B$*) if there is a bijection* $f : [C]^{2n} \to [C]^{2n}$ *such that for all possible random choices* $\tau \in [C]^{2n}$*, it holds that*

$$\ell_1^A(\tau) \geq \ell_1^B(f(\tau)),$$

where $\ell_1^A(\tau)$ *and* $\ell_1^B(f(\tau))$ *are the loads of the most loaded bins in* A *and* B *respectively with the random choices specified by* τ *and* $f(\tau)$ *respectively.*

It is immediate that the following holds.

Observation 9.3.12. $A \succeq B \Rightarrow A \succeq_n B$.

We now turn to the proof of Lemma 9.3.9.

First, notice that if we have an system of m identical machines, each of capacity 1, both the unmodified Greedy algorithm and the allocation algorithm $\mathcal{A}_{RLMS}$ will behave in exactly the same way - the load and the $\lfloor \text{load} \rfloor$ are the same if the capacity is 1. From [9], we know that the maximal load on any machine when allocating $n = m$ jobs (to m machines with capacity 1) with the Greedy algorithm, is $\Theta(\log \log n)$. Therefore, the maximal load when allocating $n = m$ jobs with $\mathcal{A}_{RLMS}$ is also $\Theta(\log \log n)$ in this

setting. We would like to show that the maximal load of a system with non-uniform machines of total capacity C is majorized by the maximal load of a system with C machines of capacity 1, when the allocating algorithm is $\mathcal{A}_{RLMS}$. We will show that the first system majorizes the second, and deduce the required result from Observation 9.3.12.

We restate Claim 2.4 of [110]:

Claim 9.3.13 ([110])**.** *Let P and Q be two normalized integer vectors such that $P \succeq Q$. If $i \leq j$ then $P + e_i \succeq Q + e_j$ where e_i is the i-th unit vector and $P + e_i$ and $Q + e_j$ are normalized.*

Lemma 9.3.14. *For allocation algorithm $\mathcal{A}_{RLMS}$, let A be a system with non-uniform machines of total capacity C, and B be a system with C uniform machines of capacity 1 each. Then $B \succeq A$.*

Proof. We use the slot load vectors of systems A and B (in B the load vector and slot load vector are identical), and show that $S^B(f(\tau)) \succeq S^A(\tau)$. The bijection is such that the jobs in both processes choose the same $k_1 < k_2 \in \{1, \ldots, C\}$ in the normalized slot load vectors, and the choice corresponds to machines k_1, k_2 in B and the machines associated with those specific slots in system A. We use induction: for $t = 0$, the claim is trivially true.

From the inductive hypothesis, before the allocation of the t-th job, $S^B_{t-1}(f(\tau)) \succeq S^A_{t-1}(\tau)$. In system B, the t-th job goes to machine k_2. In system A, if the $\lfloor \text{load} \rfloor$ of the machine of k_1 is greater than that of the machine of k_2, the job goes to k_2 if k_2 is a light slot, or to a slot to the right of k_2 (a lighter slot of the same machine), if k_2 is a heavy slot. If the $\lfloor \text{loads} \rfloor$ of the machines of k_1 and k_2 are the same, again, the job goes to k_2 if k_2 is a light slot, or to a slot to the right of k_2 (again, a lighter slot of the same machine), if k_2 is a heavy slot. In all cases, by Claim 9.3.13, it follows that $S^B(f(\tau)) \succeq S^A(\tau)$. □

Putting everything together gives our main result for this subsection.

Theorem 9.3.15. *There exists an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-local mechanism for scheduling on related machines in the restricted setting that is universally truthful and gives an $O(\log \log n)$-approximation to the makespan.*

Chapter 10

Combinatorial Auctions

Combinatorial auctions are an extremely well-studied problem in algorithmic game theory, see e.g., [25] for a survey. The general premise is the following: we wish to allocate m (possibly different) goods to n players, who have valuations for subsets of goods, with the goal of maximizing the social welfare (sometimes we may have other goals, but we focus on social welfare maximization in this work). The general problem, where each player may have an an arbitrary valuation for each subset of the goods is known to be NP-hard; indeed, even approximating the optimal solution for single-minded bidders to within $m^{1/2-\epsilon}$ is NP-hard [57]. Therefore, in order to obtain useful approximation algorithms, we must relax some of our demands. There are hundreds, if not thousands of papers devoted to various relaxations; the one we focus on is *single-minded bidders*: There are m indivisible goods, and n buyers. Each buyer i is interested in a specific subset of the items, S_i^*, and has a utility v_i if she receives all of the items in her subset and 0 otherwise. We would like find an allocation that maximizes the social welfare. Unfortunately, this problem is also NP-hard, as is approximating the social welfare to within $m^{1/2-\epsilon}$ [80]. When all buyers are interested in at most k items (which we call the *k-single minded case*), we can find a $\frac{2(k+1)}{3}$ approximation to the optimal welfare by a reduction to weighted set packing and the algorithm of Chandra and Halldórsson [19]. We can convert this approximation algorithm into a truthful mechanism by computing VCG-like payments, in the same fashion that we show in Sections 10.1.2 and 10.2, by running the allocation algorithm once again without each agent. Before tackling the k-single minded case (Section 10.2), we find local mechanisms for two simpler problems involving unit-demand buyers.

10.1 Unit-Demand Buyers

We propose local truthful mechanisms for auctions with unit-demand buyers, in which each buyer is interested in at most k items, chosen uniformly at random. First, we tackle the case where all buyers have the same valuation for the items in their sets, and the buyers' private information is their sets. Then we examine the case in which the buyers' sets are public knowledge and the buyers' private information is their valuations for their items, with the restriction that buyers have the same valuation for all items in their set.

In both cases, we use LCAs for maximal matching to obtain a $\frac{1}{2}$-approximation to the optimal solution, as any maximal matching is a $\frac{1}{2}$-approximation to the maximum matching. We could use a $(1-\epsilon)$-approximation algorithm (such as the one of Chapter 5) to obtain a better approximation ratio, but it is not clear how to guarantee truthfulness in that case. Designing a local computation mechanism that obtains an approximation ratio better than $\frac{1}{2}$ is an open problem.

10.1.1 Unit-Demand Buyers with Uniform Value

Consider the following scenario. There is a set $\mathcal{I}$ of n unit-demand buyers, and a set $\mathcal{J}$ of m indivisible items. There is a fixed, identical value for all items, which we normalize to 1. Each buyer i is interested in a set J_i of at most k items (where k is a constant). We can treat this auction as a graph $G = (V, E)$, in which $V = \mathcal{J} \cup \mathcal{I}$, and $E = \{(i, j) : i \in \mathcal{I}, j \in J_i\}$. The value of a subset S to buyer i ($v_i(S)$) is 1 if $S \cap J_i \neq \emptyset$ and 0 otherwise. Namely, the buyers are indifferent between the items in their set (they all have the same valuation for the items in their set, and a zero valuation for all other items). The utility of buyer i is quasi-linear, that is, when she receives items S and pays p her utility is $u_i(S, p) = v_i(S) - p$. We assume that the subsets J_i are selected uniformly at random and that $kn/m = O(1)$. [1]

Our goal is to design a local mechanism that maximizes the social welfare. In order to do this, we would like to satisfy as many buyers as possible, allocating each buyer a single item from her set. We call this type of auction an $k-$UDUV (unit demand, uniform value) auction.

Ideally, we would like to find a maximum matching between the buyers and items,

[1] Hence, as in Claim 2.3.5, the graph is $(n/2)$-conditionally d-light, for some constant d.

as this will maximize the social welfare. However, Corollary 2.5.2 shows that it not possible to solve the maximum matching problem locally, even on bipartite graphs. We will therefore content ourselves with finding an *approximation* to the maximum matching.

10.1.1.1 A $\frac{1}{2}$-approximation to the maximum matching

To obtain a $\frac{1}{2}$-approximation, we use the $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA for finding maximal matchings in undirected graphs of Chapter 4. We denote this algorithm by $\mathcal{A}_{UDUV}$.

Our mechanism $\mathcal{M}_{UDUV} = (\mathcal{A}_{UDUV}, \mathcal{P}_{UDUV})$ works as follows. The mechanism receives from each buyer i a subset $J'_i \subset J$. As discussed previously, $\mathcal{A}_{UDUV}$ generates a random order in which it considers the items; clearly the buyers cannot influence the order in which the items as considered.

As the values of all items are identical, any payment scheme $\mathcal{P}_{UDUV}$ that fulfills *voluntary participation* is adequate, i.e., the payment can be any value in the range $[0, 1]$. For example, charge $p = 1/2$ from any buyer that receives an item and $p = 0$ from any buyer that does not receive an item.

In order to show that our mechanism is truthful, we need only to show that buyers cannot profit by bidding $J'_i \neq J_i$.

Theorem 10.1.1. *In the k-UDUV auction, the mechanism $\mathcal{M}_{UDUV} = (\mathcal{A}_{UDUV}, \mathcal{P}_{UDUV})$ is universally truthful and provides a $\frac{1}{2}$-approximation to the optimal allocation.*

Proof. The proof will be done in two steps. First, we show that for any J'_i, bidding $J_i \cap J'_i$ weakly dominates bidding J'_i. Second, we show that bidding J_i weakly dominates bidding any $J^*_i \subseteq J_i$.

To show that $J_i \cap J'_i$ weakly dominates bidding J'_i, label the items in $J_i \cap J'_i$ as *good*, and those in $J'_i \setminus J_i$ by *bad*. If a good item is allocated to buyer i when she bids J'_i, it will also be allocated to her when bidding $J_i \cap J'_i$. Therefore the value of buyer i cannot decrease by bidding $J_i \cap J'_i$, and hence $J_i \cap J'_i$ weakly dominates bidding J'_i.

To show that bidding J_i weakly dominates bidding any $J^*_i \subseteq J_i$, consider the following. If buyer i does not receive any items when bidding J^*_i, the claim trivially holds. Assume buyer i receives item j when bidding J^*_i. Then, when bidding J_i, if she

has not received any item from $J_i \setminus J_i^*$ before considering item j, then she will receive item j. Therefore, if she receives an item when bidding J_i^*, she will also receive an item when bidding J_i, and have the same valuation and utility. This proves that bidding J_i weakly dominates bidding $J_i^* \subseteq J_i$.

The reasoning that the allocation is a $\frac{1}{2}$-approximation is similar to the proof of maximal versus maximum matching. Consider a buyer that is not allocated an item in $\mathcal{A}_{UDUV}$ and *is* allocated an item in the optimal allocation. Her item is allocated to a unique different buyer in $\mathcal{A}_{UDUV}$. This bounds the number of buyers allocated items in the optimal allocation and not in $\mathcal{A}_{UDUV}$ by the number of buyers that are allocated items in $\mathcal{A}_{UDUV}$, giving the factor of $\frac{1}{2}$ approximation, and completing the proof of the theorem. □

Theorem 10.1.2. *The k-UDUV auction has an* $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$*-local mechanism that is universally truthful and provides a* $\frac{1}{2}$*-approximation to the optimal social welfare.*

10.1.2 Unit Demand Buyers, Uniform-Buyer-Value

There is a set $\mathcal{I}$ of n buyers, and a set $\mathcal{J}$ of m items. Each buyer i is interested in a set of at most k items, $J_i \subseteq \mathcal{J}$, which is chosen uniformly at random from all possible sets, is public knowledge, and has a private valuation, t_i (which represents the value of any item from J_i to buyer i). Buyer i's valuation for subset S is $v_i(S) = t_i$ if $S \cap J_i \neq \emptyset$, and 0 otherwise. The utility of buyer i is quasi-linear, namely her utility of receiving subset S and paying p is $u_i(S, p) = v_i(S) - p$.

Similarly to Claim 2.3.5, we can treat this auction as a ((polylog n)-conditionally d-light) weighted graph $G = (V, E)$, in which $V = \mathcal{J} \cup \mathcal{I}$, and $E = \cup_i E_i$ where $E_i = \{(i, j) : j \in J_i\}$. Every edge $e \in E_i$ has weight $w(e) = t_i$.

In addition, we make the simplifying assumption that the buyers are Bayesian - the valuations t_i are randomly drawn from some prior (not necessarily known) distribution, that is identical to all buyers. We call this type of auction an $k-$UDUBV (unit demand, uniform buyer value) auction.

We require that if buyer i does not receive an item, she pays nothing. If buyer i receives an item, the mechanism charges her $p_i(b)$, where b is the bid vector. Any buyer will receive at most one item in the allocation of the mechanism. We would like

to ensure that bidding truthfully is a dominant strategy for all buyers. Hence, we need to show that, for all b_i and b_{-i}, we have $u_i(t_i, b_{-i}) \geq u_i(b_i, b_{-i})$.

The allocation algorithm, $\mathcal{A}_{UDUBV}$, is as follows. First, $\mathcal{A}_{UDUBV}$ orders the buyers by their bids. Starting with the buyer with the highest bid, each buyer i is allocated an item $j_i \in J_i$ such that j_i has not yet been allocated. If more than one such item exists, we allocate the (lexicographically) first $j_i \in J_i$. (We assume the items have lexicographic order.) If there is no such item, then buyer i is not allocated any item. We continue until we cannot allocate any more items.

First, we claim that the resulting allocation is a $\frac{1}{2}$-approximation.

Claim 10.1.3. *The allocation algorithm $\mathcal{A}_{UDUBV}$ provides a $\frac{1}{2}$-approximation to the optimal allocation, with respect to the bids b.*

Proof. The proof is similar to the proof that any maximal matching is a $\frac{1}{2}$-approximation to a maximum matching. Regard the auction as a bipartite graph $G = (U, W, E)$, with U representing the buyers and W representing the items. There is a weighted edge between each buyer i and every item $j_i \in J_i$. The weight of each edge $e = (i, j_i)$ is the bid of buyer i, b_i. The optimal allocation is a maximum weighted matching, while $\mathcal{A}_{UDUBV}$ considers the buyers in the order of their b_i's and finds a maximal matching.

If an edge $e = (i, j_i)$ is added in $\mathcal{A}_{UDUBV}$ but not in the optimal matching, then it is allocated instead of at most 2 edges in the optimal matching (an edge e' containing i and an edge e'' containing j_i). Because $\mathcal{A}_{UDUBV}$ considers edges according to their weights, we know that $w(e) \geq w(e')$ and $w(e) \geq w(e'')$. Therefore $2w(e) \geq w(e') + w(e'')$ and so the ratio between the allocation of $\mathcal{A}_{UDUBV}$ and the optimal allocation is at least $\frac{1}{2}$. □

We now need to specify the payment mechanism. To calculate buyer i's payment when she receives an item, we run $\mathcal{A}_{UDUBV}$ without buyer i. Buyer i pays the smallest value for which any of her items is sold when the auction is run without her. (This is exactly the minimal value of b_i which would still gain her an item). We label buyer i's payment by p_i, hence, the payments are $\mathcal{P}_{UDUBV} = \{p_1, \ldots p_n\}$.

Claim 10.1.4. *In mechanism $\mathcal{M}_{UDUBV} = (\mathcal{A}_{UDUBV}, \mathcal{P}_{UDUBV})$, for all buyers i and all b_i, bidding t_i weakly dominates bidding b_i.*

Proof. We will show that, fixing the bids of all other buyers at b_{-i},

1. Buyer i has no incentive to over-bid, i.e., bid $b_i > t_i$.

2. Buyer i has no incentive to under-bid i.e., bid $b_i < t_i$.

To prove (1), we notice that if buyer i receives an item, then she has no incentive to bid higher, as she has no preference between items. Furthermore, bidding higher cannot change her payment, as her payment is independent of her bid. If she does not receive an item, then $p_i \geq b_i (= t_i)$[2], and so if she bids more, she might receive an item, but will have to pay at least t_i if she does, which will result in a non-positive utility.

To prove (2), we notice that if buyer i does not receive an item, she cannot obtain an item by bidding lower, because the algorithm allocates first to higher bids. If she is allocated an item, then bidding lower will not make a difference, unless she bids under p_i, in which case she will not receive any item, and hence have zero utility. □

Claims 10.1.3 and 10.1.4 imply the following.

Theorem 10.1.5. *The mechanism $\mathcal{M}_{UDUBV}$ is universally truthful and provides a $\frac{1}{2}$-approximation to the optimal social welfare.*

Algorithm $\mathcal{A}_{UDUBV}$ is a $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-LCA for maximal matching on $(\text{polylog}\, n)$-conditionally d-light graphs. Notice, however, that we need to run $\mathcal{A}_{UDUBV}$ once for calculating the allocation, and k more times for calculating the payment. Hence, we have the following.

Theorem 10.1.6. *There is an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-local mechanism for $k-UDUBV$ auction that is universally truthful and provides a $\frac{1}{2}$-approximation to the optimal social welfare.*

10.2 k- Single Minded Bidders

We extend the results of Section 10.1.2 to the case of combinatorial auctions with k-single-minded bidders: There is a set $\mathcal{I}$ of n buyers, and a set $\mathcal{J}$ of m items. Each buyer i is interested in a set of at most k items, $J_i \subseteq \mathcal{J}$, which is public knowledge, and has a private valuation, t_i, which represents the value of the entire subset J_i to

[2] As specified, if buyer i does not receive an item, she pays 0. However, if the mechanism were to compute the payment, i.e., run the mechanism without her, the payment would be $p_i \geq b_i (= t_i)$.

buyer i. Buyer i's valuation for subset S is $v_i(S) = t_i$ if $J_i \subseteq S$, and 0 otherwise. The utility of buyer i is quasi-linear, namely her utility of receiving subset S and paying p is $u_i(S, p) = v_i(S) - p$.

As in Subsection 10.1.2, we assume that J_i has a uniform or binomial distribution, and that the valuations t_i are randomly drawn from some prior (not necessarily known) distribution, and $kn/m = O(1)$.

The allocation algorithm, $\mathcal{A}_{kSMB}$, is as follows. First, $\mathcal{A}_{kSMB}$ orders the buyers by their bids. Starting with the buyer with the highest bid, each buyer i is allocated subset J_i such that no item $j_i \in J_i$ has been allocated yet. We continue until we cannot allocate any more subsets.

Claim 10.2.1. *The allocation algorithm $\mathcal{A}_{kSMB}$ provides a $\frac{1}{k}$-approximation to the optimal allocation, with respect to the values b.*

Proof. Compare the allocation of Algorithm $\mathcal{A}_{kSMB}$, J^*, to the optimal allocation, OPT. Each set $J \in J^*$ is chosen by $\mathcal{A}_{kSMB}$ instead of at most k sets in OPT, but its weight is greater than each of their weights, because $\mathcal{A}_{kSMB}$ is a greedy algorithm. □

The payment scheme is as follows. To calculate buyer i's payment when she receives an item, we run $\mathcal{A}_{kSMB}$ without buyer i. Buyer i pays the highest value of the allocated sets J_x for which $J_i \cap J_x \neq \emptyset$. (This is exactly the minimal value of b_i which would still gain her an item). We label buyer i's payment by p_i, and let $\mathcal{P}_{kSMB} = \{p_1, \ldots p_n\}$.

Claim 10.2.2. *In mechanism $\mathcal{M}_{kSMB} = (\mathcal{A}_{kSMB}, \mathcal{P}_{kSMB})$, for all buyers i and all b_i, bidding t_i weakly dominates bidding b_i.*

The proof is similar to the proof of Claim 10.1.4 and is omitted.

Combining Claims 10.2.1 and 10.2.2, we get

Theorem 10.2.3. *There exists an $(O(\log n), O(\log^2 n), O(\log n), O(\log^2 n), 1/n)$-local mechanism for combinatorial auctions with known $k-$single minded bidders (where the sets are sampled uniformly at random) that is universally truthful and provides a $\frac{1}{k}$-approximation to the optimal social welfare.*

Chapter 11

Discussion and Future Directions

In the first part of the thesis we described several techniques for designing local computation algorithms; the two main ones being (1) reductions to distributed (and parallel) algorithms and (2) reductions to online algorithms. The reduction to distributed algorithms is due of Parnas and Ron; our main result was proving a tighter bound for the running time of the simulation of the distributed algorithm on d-light graphs. For the reduction to online algorithms, we showed that we can use a random seed of length $O(\log n)$ to generate a random ranking the vertices (or edges) of the graph, and that we can simulate an online algorithm on the vertices (edges) in the order defined by this ranking. This allows us to obtain LCAs that require time $O(\log^2 n)$ and transient space $O(\log n \log\log n)$ or vice versa. Given a truly random ordering on the vertices of a line graph, with probability at least $\frac{1}{n}$, there will be a monotone increasing segment of length $\Omega(\frac{\log n}{\log\log n})$. This gives an immediate lower bound on the time and transient space requirements of our LCAs, when we apply this reduction. It would be interesting to obtain tighter bounds. Furthermore, it would be interesting to extend the techniques of the thesis (and others), to obtain better bounds for similar LCAs on a wider family of graphs.

In some cases (for example in our reduction to the online greedy MIS algorithm), we can use a known distributed or online algorithm as a black box; in other cases, we may benefit from designing new (or variations on known) distributed, parallel and online algorithms. An interesting example of a new parallel algorithm that was designed especially for the purpose of reduction from LCAs is the maximum weight forest algorithm of Chapter 6. There are many known parallel algorithms for maximum weight forests, and they all outperform our new algorithm, because it (the new one) performs many

redundant operations. However, this "flaw" is exactly what makes it simple to adapt it to an LCA. A research direction for future work is to develop new parallel and online algorithms to tackle problems that the current reductions do not hold for.

Most of the work on LCAs thus far has focused on proving upper bounds for specific (families of) algorithms on specific (families of) graphs. Other than result presented in Chapter 2, we are not aware of any impossibility results. One of the more important directions in the study of local computation algorithms seems to be the lower bounds and more impossibility results, in order to better characterize the possibilities and limits of LCAs.

In the second part of the thesis we designed local computation mechanisms for several game theoretic problems. We showed how we can adapt our maximal matching LCA to obtain a local mechanism for the housing allocation problem. We then tackled the more complex problem of stable matching, and described an LCA for it based on the Gale-Shapley algorithm. We showed that, when the men's preference lists are random and have bounded length, there is an LCA that finds an "almost stable" matching. Interestingly, we were able to use our techniques to obtain a non-local result: we showed that in the general case when the men's list are bounded, stopping the Gale-Shapley algorithm after a constant number of rounds can guarantee an arbitrarily stable matching. To our knowledge, this was the first time techniques from LCAs were used to obtain results in other fields. Others have also used LCAs to design and analyze algorithms that are not LCAs; for example, Even et al. [30] used LCAs to design better distributed algorithms for maximum matching. It would be interesting to continue finding links between LCAs and other fields, and to see whether the techniques developed for LCAs will be useful in other disciplines as well.

Finally, we showed local computation mechanisms for two load balancing problems, two combinatorial auctions with unit-demand bidders and combinatorial auctions with k-minded bidders. We feel that this is the first step in the field or local computation mechanism design, which may become a more practical field as markets continue to grow.

Bibliography

[1] The size of the World Wide Web (the Internet), 2015. Online; accessed 25-May-2015. http://www.worldwidewebsize.com. 1

[2] Nir Ailon, Bernard Chazelle, Seshadhri Comandur, and Ding Liu. Property-preserving data reconstruction. *Algorithmica*, 51(2):160–182, 2008. 1.1

[3] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple constructions of almost k-wise independent random variables. In *FOCS*, pages 544–553, 1990. 1.1.3, 1.1.3

[4] Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proc. 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1132–1139, 2012. 1.1.2, 1.1.3, 1.1.4, 1.4, 4, 4.2

[5] Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, 3rd edition, 2008. 8.1.4.1, A.1.2

[6] R. Andersen, C. Borgs, J. Chayes, J. Hopcroft, V. Mirrokni, and S. Teng. Local computation of pagerank contributions. *Internet Mathematics*, 5(1–2):23–45, 2008. 1.1

[7] Aaron Archer, Christos H. Papadimitriou, Kunal Talwar, and Éva Tardos. An approximate truthful mechanism for combinatorial auctions with single parameter agents. *Internet Mathematics*, 1(2), 2003. 9.2

[8] Aaron Archer and Éva Tardos. Truthful mechanisms for one-parameter agents. In *Proc. 42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, 2001. 9, 9.1, 9.2, 9.3

[9] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999. 3.1, 9, 9.1, 9.3

[10] Yossi Azar, Joseph Naor, and Raphael Rom. The competitiveness of on-line assignments. *J. Algorithms*, 18(2):221–237, 1995. 9.1

[11] Moshe Babaioff, Robert D. Kleinberg, and Aleksandrs Slivkins. Truthful mechanisms with implicit payment computation. In *ACM Conference on Electronic Commerce*, pages 43–52, 2010. 9.2

[12] Jérémy Barbay and Gonzalo Navarro. Compressed representations of permutations, and applications. In *26th International Symposium on Theoretical Aspects of Computer Science, STACS*, pages 111–122, 2009. 1.1.3

[13] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. *SIAM J. Comput.*, 35(6):1350–1385, 2006. 9.1.4

[14] Petra Berenbrink, André Brinkmann, Tom Friedetzky, and Lars Nagel. Balls into non-uniform bins. *J. Parallel Distrib. Comput.*, 74(2):2065–2076, 2014. 3.1, 9, 9.1, 9.1.6, 9.2, 9.2, 9.2, 9.3

[15] Pavel Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3:41–62, 2006. 1.1

[16] R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37:194–197, 1941. 1

[17] John W. Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In *IPTPS*, pages 80–87, 2003. 9.1.7, 9.2

[18] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979. 1.1.3

[19] Barun Chandra and Magnús M. Halldórsson. Greedy local improvement and weighted set packing approximation. *J. Algorithms*, 39(2):223–240, 2001. 10

[20] George Christodoulou and Annamária Kovács. A deterministic truthful ptas for scheduling related machines. In *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1005–1016, 2010. 9

[21] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32 – 53, 1986. 1.1.5

[22] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, 1971. 1

[23] Artur Czumaj and Christian Sohler. Sublinear-time algorithms. In *Property Testing - Current Research and Surveys*, pages 41–64, 2010. 1.1.1

[24] Andrzej Czygrinow and Michal Hanckowiak. Distributed algorithm for better approximation of the maximum matching. In *COCOON*, pages 242–251, 2003. 5.1

[25] Sven de Vries and Rakesh V. Vohra. Combinatorial auctions: A survey. *INFORMS J. on Computing*, 15(3):284–309, 2003. 10

[26] Jack Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17:449–467, 1965. 2.5

[27] Kimmo Eriksson and Olle Häggström. Instability of matchings in decentralized markets with various preference structures. *Int. J. Game Theory*, 36(3-4):409–420, 2008. 8

[28] Guy Even, Moti Medina, and Dana Ron. Deterministic stateless centralized local algorithms for bounded degree graphs. In *22th Annual European Symposium on Algorithms (ESA)*, pages 394–405, 2014. 1.1.4, 1.1.5

[29] Guy Even, Moti Medina, and Dana Ron. Distributed maximum matching in bounded degree graphs. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015,*, pages 18:1–18:10, 2015. 1.1, 5, 5.1, 6

[30] Guy Even, Moti Medina, and Dana Ron. Distributed maximum matching in bounded degree graphs. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN*, pages 18:1–18:10, 2015. 11

[31] Tomás Feder, Nimrod Megiddo, and Serge A. Plotkin. A sublinear parallel algorithm for stable matching. *Theor. Comput. Sci.*, 233(1-2):297–308, 2000. 8

[32] Tamás Fleiner. A fixed-point approach to stable matchings and some applications. *Math. Oper. Res.*, 28(1):103–126, February 2003. 8.1

[33] Patrik Floréen, Petteri Kaski, Valentin Polishchuk, and Jukka Suomela. Almost stable matchings by truncating the gale-shapley algorithm. *Algorithmica*, 58(1):102–118, 2010. 8, 8.1

[34] David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–14, 1962. 1.3.2, 8

[35] David Gale and Marilda Sotomayor. Some remarks on the stable matching problem. *Discrete Applied Mathematics*, 11(3):223 – 232, 1985. 8.1

[36] N. Garg, V.V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997. (document), 6.2, 12

[37] Oded Goldreich. A brief introduction to property testing. In *Studies in Complexity and Cryptography*, pages 465–469, 2011. 1

[38] Mika Göös, Juho Hirvonen, and Jukka Suomela. Lower bounds for local approximation. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 175–184, 2012. 1.1.5

[39] Dan Gusfield and Robert W. Irving. *The Stable marriage problem - structure and algorithms*. Foundations of computing series. MIT Press, 1989. 8

[40] Philip Hall. On representatives of subsets. *J. London Math. Soc.*, 10(1):26–30, 1935. 1

[41] Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIAM J. Discrete Math.*, 15(1):41–57, 2001. 3.1

[42] Avinatan Hassidim, Yishay Mansour, and Shai Vardi. Local computation mechanism design. In *ACM Conference on Economics and Computation, EC '14*, pages 601–616, 2014. To appear in *Transactions on Economics and Computation*. 1.1.4, 1.4, 4.2, 7

[43] Martin Hilbert and Priscila López. The worlds technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 1 April 2011. 1

[44] Dorit S. Hochbaum and David B. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing*, 17(3):539–551, 1988. 9

[45] Jaap-Henk Hoepman, Shay Kutten, and Zvi Lotker. Efficient distributed weighted matchings on trees. In *SIROCCO*, pages 115–129, 2006. 5.1

[46] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. 5.1, 5.2, 5.2.1, 5.2.2, 1

[47] Nicole Immorlica and Mohammad Mahdian. Marriage, honesty, and stability. In *SODA*, pages 53–62, 2005. 3.1, 8

[48] Amos Israeli and Alon Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):77–80, 1986. 5.1, 6.3.3

[49] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 271–279, New York, NY, USA, 2003. ACM. 1.1

[50] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972. 1

[51] J. Katz and L. Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *Proc. 32nd Annual ACM Symposium on the Theory of Computing (STOC)*, pages 80–86, 2000. 1.1

[52] Donald E. Knuth. Mariages stables. *Les Presses de l'Université de Montréal*, 1976. 8.1.4.1

[53] Fuhito Kojima and Parag A. Pathak. Incentives and stability in large two-sided matching markets. *American Economic Review*, 99(3):608 –627, 2009. 3.1, 8

[54] Fabian Kuhn. Local approximation of covering and packing problems. In *Encyclopedia of Algorithms*. 2008. 1.1.5

[55] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 980–989, 2006. 1.1.1, 1.1.5, 5.1

[56] H. O. Lancaster. Pairwise statistical independence. *Annals of Mathematical Statistics*, 36:1313–1317, 1965. 1.1.3

[57] Daniel J. Lehmann, Liadan O'Callaghan, and Yoav Shoham. Truth revelation in approximately efficient combinatorial auctions. *J. ACM*, 49(5):577–602, 2002. 10

[58] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. In *Proc. 28th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 217–224, 1987. 9

[59] Christoph Lenzen and Roger Wattenhofer. Leveraging Linial's locality limit. In *Distributed Computing, 22nd International Symposium, DISC*, pages 394–407, 2008. 6.3.3

[60] Reut Levi, Dana Ron, and Ronitt Rubinfeld. Local algorithms for sparse spanning graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, pages 826–842, 2014. 1.1.4

[61] Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1), 1992. 1.1.5

[62] Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved distributed approximate matching. In *Proc. 20th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 129–136, 2008. 5.1, 5.2, 5.2.3, 5.2

[63] Zvi Lotker, Boaz Patt-Shamir, and Adi Rosén. Distributed approximate matching. *SIAM J. Comput.*, 39(2), 2009. 6.3

[64] Enyue Lu and S. Q. Zheng. A parallel iterative improvement stable matching algorithm. In *HiPC*, pages 55–65, 2003. 8

[65] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986. 1.1.1, 1.1.3

[66] Yishay Mansour, Boaz Patt-Shamir, and Shai Vardi. Constant-time local computation algorithms. In *Approximation and Online Algorithms - 13th International Workshop, WAOA*, pages 110–121, 2015. 1.1.5, 1.4, 6

[67] Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. Converting online algorithms to local computation algorithms. In *Proc. 39th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 653–664, 2012. 1.1.2, 1.1.4, 1.4, 4, 4.2

[68] Yishay Mansour and Shai Vardi. A local computation approximation scheme to maximum matching. In *APPROX-RANDOM*, pages 260–273, 2013. 1.1.4, 1.4, 4.2, 5

[69] S. Marko and D. Ron. Distance approximation in bounded-degree and general sparse graphs. In *APPROX-RANDOM'06*, pages 475–486, 2006. 1.1.1

[70] Michael Maschler, Eilon Solan, and Shmuel Zamir. *Game Theory*. Cambridge Press, 2013. 8

[71] Ueli Maurer and Krzysztof Pietrzak. Composition of random systems: When two weak make one strong. In Moni Naor, editor, *Theory of Cryptography*, volume 2951 of *Lecture Notes in Computer Science*, pages 410–427. Springer Berlin Heidelberg, 2004. 4.1.1, 4.1.6

[72] Raghu Meka, Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Fast pseudorandomness for independence and load balancing - (extended abstract). In *ICALP (1)*, pages 859–870, 2014. 1.1.3, 4.1.1, 4.1.8

[73] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005. 1

[74] Roger B. Myerson. Optimal auction desing. *Mathematics of Operations Research*, 6:58–74, 1981. 9.1, 9.1.2

[75] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. In *STOC*, pages 213–223, 1990. 1.1.3, 4.1.1

[76] Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. 1.1.5

[77] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001. 6.1

[78] Cheng Ng and Daniel S. Hirschberg. Lower bounds for the stable marriage problem and its variants. *SIAM J. Comput.*, 19(1):71–77, 1990. 8

[79] Huy N. Nguyen and Krzystof Onak. Constant-time approximation algorithms via local improvements. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 327–336, 2008. (document), 1, 1.1.1, 1.1.2, 6.3.3

[80] N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani, editors. *Algorithmic Game Theory*. Cambridge University Press, 2005. 1.2, 7.1, 10

[81] Noam Nisan and Amir Ronen. Algorithmic mechanism design (extended abstract). In *Proc. 31st Annual ACM Symposium on the Theory of Computing (STOC)*, pages 129–140, 1999. 1.2, 9

[82] M. Shaked ond J.G. Shanthikumar. *Stochastic Orders and their applications*. Academic Press, Inc., San Diego, CA, USA, 1994. A.2

[83] Rafail Ostrovsky and Will Rosenbaum. On the communication complexity of finding an (approximate) stable marriage. *CoRR*, abs/1406.1273, 2014. 8

[84] James Oxley. *Matroid Theory,*. Oxford University Press, 1992. 6.1

[85] Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001. 1.1.5

[86] M. Parnas and D. Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theoretical Computer Science*, 381(1–3), 2007. (document), 1, 1.1.1, 1.1.1, 3, 3.1

[87] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. 1.1.5

[88] Michael J. Quinn. A note on two parallel algorithms to solve the stable marriage ppoblem. *BIT Numerical Mathematics*, 25(3):473–476, 1985. 8

[89] Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. 1.1.1

[90] Omer Reingold and Shai Vardi. New techniques and tighter bounds for local computation algorithms. *J. Comput. Syst. Sci.*, 82(7):1180–1200, 2016. 1.1.2, 1.1.3, 1.1.4, 1.4, 4

[91] Dana Ron. Algorithmic and analysis techniques in property testing. *Foundations and Trends in Theoretical Computer Science*, 5(2):73–205, 2009. 1

[92] Alvin E. Roth. The origins, history, and design of the resident match. *Journal of the American Medical Association*, 289(7):909–912, 2003. 8

[93] Alvin E. Roth and Elliott Peranson. The redesign of the matching market for american physicians: Some engineering aspects of economic design. *American Economic Review*, 89:748–780, 1999. 8

[94] Alvin E. Roth and Uriel G. Rothblum. Truncation strategies in matching markets – in search of advice for participants. *Econometrica*, 67(1):21–43, 1999. 8

[95] Alvin E. Roth and Marilda Sotomayor. *Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis*. Cambridge University Press, 1990. 8

[96] Alvin E. Roth and Xiaolin Xing. Turnaround time and bottlenecks in market clearing: Decentralized matching in the market for clinical psychologists. *Journal of Politial Economy*, 105:284–329, 1997. 8

[97] Ronitt Rubinfeld and Asaf Shapira. Sublinear time algorithms. *SIAM J. Discrete Math.*, 25(4):1562–1588, 2011. 1, 1.1.1

[98] Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. In *Proc. 2nd Symposium on Innovations in Computer Science (ICS)*, pages 223–238, 2011. 1.1, 1.1.1, 1.1.4

[99] M. E. Saks and C. Seshadhri. Local monotonicity reconstruction. *SIAM Journal on Computing*, 39(7):2897–2926, 2010. 1.1

[100] Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24, 2013. 1.1.5

[101] Lars-Gunnar Svensson. Strategy-proof allocation of indivisible goods. *Social Choice and Welfare*, 16(4):557–567, 1999. 7.2, 2

[102] Kunal Talwar and Udi Wieder. Balanced allocations: the weighted case. In *Proc. 39th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 256–265, 2007. 9

[103] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983. 6.1.1, 6.1.4

[104] S. S. Tseng and Richard C. T. Lee. A parallel algorithm to solve the stable marriage problem. *BIT*, 24(3):308–316, 1984. 8

[105] Ryuhei Uehara and Zhi-Zhong Chen. Parallel approximation algorithms for maximum weighted matching in general graphs. In *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS*, pages 84–98, 2000. 6.3

[106] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1–3):1–336, 2011. 1.1.3, 4.1.7

[107] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001. (document), 1, 12, 6.2.1

[108] Berthold Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4):568–589, 2003. 3.1, 9, 9.1.5

[109] Mirjam Wattenhofer and Roger Wattenhofer. Distributed weighted matching. In *DISC*, pages 335–348, 2004. 6.3

[110] Udi Wieder. Balanced allocations with heterogenous balls. In *Proc. 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 188–193, 2007. 9.1, 9.2, 9.3, 9.3, 9.3.13

[111] Wikipedia. Facebook — Wikipedia, the free encyclopedia, 2015. Online; accessed 25-May-2015. http://en.wikipedia.org/wiki/Facebook. 1

[112] Yuichi Yoshida, Masaki Yamamoto, and Hiro Ito. Improved constant-time approximation algorithms for maximum matchings and other optimization problems. *SIAM J. Comput.*, 41(4):1074–1093, 2012. 1.1.1

[113] Jian Zhang. A survey on streaming algorithms for massive graphs. In *Managing and Mining Graph Data*, pages 393–420. Springer, 2010. 1

Appendix A

Auxiliary Results

A.1 Chernoff Bounds

Theorem A.1.1 (Chernoff bound)**.** *Let X be a binomially distributed random variable, $X \sim B(n,p)$, such that $\mu = np$. Then, for $\lambda > 2e-1$ it holds that*

$$\Pr[X > (1+\lambda)\mu] < 2^{-\mu\lambda}.$$

Proof. Substituting $\lambda \geq 2e-1$ into the standard Chernoff bound gives

$$\begin{aligned}\Pr[X > (1+\lambda)\mu] &\leq \left(\frac{e^{\lambda}}{(1+\lambda)^{1+\lambda}}\right)^{\mu} \\ &\leq \left(\frac{e^{\lambda}}{(2e)^{1+\lambda}}\right)^{\mu} \\ &=\leq 2^{-\mu\lambda}\end{aligned}$$

□

Lemma A.1.2 (Azuma's Inequality [5])**.** *Let $c = Y_0, \ldots, Y_n$ be a martingale with $|Y_{i+1} - Y_i| \leq 1$ for all $0 \leq i \leq n$. Then*

$$\Pr[|Y_n - c| > \lambda\sqrt{n}] < 2e^{-\lambda^2/2}.$$

A.2 Stochastic Dominance

We recall the following facts about stochastic dominance (see, e.g., [82]).

1. If $X \leq_{st} Y$ and $Y \leq_{st} Z$ then $X \leq_{st} Z$.
2. For any integer n, let $\{X_1, X_2, \ldots X_n\}$ and $\{Y_1, Y_2, \ldots Y_n\}$ be two sequences of independent random variables. If $\forall i, X_i \leq_{st} Y_i$, then $\sum_{j=1}^{n} X_j \leq_{st} \sum_{j=1}^{n} Y_j$.

We need the following observation and lemmas.

Observation A.2.1. *For any $x \leq \alpha, B(x, \frac{d}{\alpha}) \leq_{st} B(\alpha, \frac{d}{\alpha})$.*

Lemma A.2.2. *Let Y_1, Y_2 be independent discrete random variables. Let X_1, X_2 be (possibly) dependent discrete random variables, such that X_1 is independent of Y_2. If $X_1 \leq_{st} Y_1$, and conditioned on any realization of X_1, it holds that $X_2 \leq_{st} Y_2$, then*

$$X_1 + X_2 \leq_{st} Y_1 + Y_2.$$

Proof. For every realization x of X_1, define a different random variable for X_2. That is $X_2(x) = X_2|X_1 = x$. Note $\forall x, X_2(x) \leq_{st} Y_2$. Further note that X_1 and Y_2 are independent. From the law of total probability,

$$\begin{aligned} Pr[X_1 + X_2 > k] &= \sum_x Pr[x + X_2(x) > k] \cdot Pr[X_1 = x] \\ &= \sum_x Pr[X_2(x) > k - x] \cdot Pr[X_1 = x] \\ &\leq \sum_x Pr[Y_2 > k - x] \cdot Pr[X_1 = x] \\ &= \sum_x Pr[x + Y_2 > k] \cdot Pr[X_1 = x] \\ &= Pr[X_1 + Y_2 > k] \\ &\leq Pr[Y_1 + Y_2 > k]. \end{aligned}$$

□

This implies

Lemma A.2.3. *Let $\{Y_1, Y_2, \ldots, Y_N\}$ be a sequence of independent random variables. Let $\{X_1, X_2, \ldots, X_N\}$ be a series of (possibly) dependent random variables. If it holds that $X_1 \leq_{st} Y_1$, and for any $1 \leq i \leq N$, conditioned on any realization of $X_1, \ldots X_{i-1}$, it holds that $X_i \leq_{st} Y_i$, then*

$$\sum_{j=1}^{N} X_j \leq_{st} \sum_{j=1}^{N} Y_j.$$

Proof. We prove by induction on i that $\sum_{j=1}^{i} X_j \leq_{st} \sum_{j=1}^{i} Y_j$. For $i = 1$ we have that $X_1 \leq_{st} Y_1$. Assume that $\sum_{j=1}^{i} X_j \leq_{st} \sum_{j=1}^{i} Y_j$, we prove that $\sum_{j=1}^{i+1} X_j \leq_{st} \sum_{j=1}^{i+1} Y_j$. Let $Z_1 = \sum_{j=1}^{i} X_j$, $Z_2 = X_{i+1}$, $W_1 = \sum_{j=1}^{i} Y_j$, and $W_2 = Y_{i+1}$. Applying Lemma A.2.2 with Z_1, Z_2, W_1, W_2 we get that $Z_1 + Z_2 \leq_{st} W_1 + W_2$ as required. □

For the second lemma we prove, we require the following well-known inequalities:

Fact A.2.4. *For every $0 < x < 1$ and every $y > 0$,*

$$\left(1 - \frac{x}{y}\right)^y < e^{-x} < 1 - \frac{x}{2}.$$

Claim A.2.5. *Let $2d < \alpha \leq n$. Let X and Y be random variables such that $X \sim B(1, \frac{d}{\alpha})$ and $Y \sim B(\lceil \frac{n^2}{\alpha} \rceil, \frac{2d}{n^2})$. Then $X \leq_{st} Y$.*

Proof. X is in fact a random variable with the Bernoulli distribution. As X can only take the values 0 or 1, to show stochastic dominance, it suffices to show that $\Pr[Y = 0] \leq \Pr[X = 0]$.

$$\Pr[Y = 0] = \left(1 - \frac{2d}{n^2}\right)^{\lceil \frac{n^2}{\alpha} \rceil} < e^{-2d/\alpha} < 1 - \frac{d}{\alpha} = \Pr[X = 0].$$

□

Claim A.2.6. *Let X be a random variable such that $X \sim B(\alpha, \frac{d}{\alpha})$, and let $X_1, X_2, \ldots, X_{\alpha-1}$ be random variables duch that $\forall i, X_i \sim B(1, \frac{d}{\alpha})$. Then $X \leq_{st} \sum_{i=1}^{\alpha-1} X_i + 1$.*

The proof is immediate from the definition of the binomial distribution.

Claim A.2.7. *Let Y be a random variable such that $Y \sim B(n^2, \frac{2d}{n^2})$ and let $Y_1, Y_2, \ldots, Y_{\alpha-1}$ be random variables such that $\forall i, Y_i \sim B(\lceil \frac{n^2}{\alpha} \rceil, \frac{2d}{n^2})$. Then $\sum_{i=1}^{\alpha-1} Y_i \leq_{st} Y$.*

Proof. It suffices to show that $(\alpha - 1)\lceil \frac{n^2}{\alpha} \rceil \leq n^2$.

$$(\alpha - 1)\left\lceil \frac{n^2}{\alpha} \right\rceil \leq (\alpha - 1)(\frac{n^2}{\alpha} + 1) \leq n^2 - n + \alpha - 1 < n^2,$$

because $\alpha \leq n$. □

Combining Claims A.2.5, A.2.6 and A.2.7, we get

Lemma A.2.8. *Let Z and X be random variables such that $Z \sim 2d + B(n^2, \frac{2d}{n^2})$ and $X \sim B(\alpha, \frac{d}{\alpha})$, where $d \leq \alpha \leq n$. Then $X \leq_{st} Z$.*

Proof. If $\alpha \leq 2d$, the lemma holds immediately. Assume $\alpha \geq 2d$. Then, using the notation of Claims A.2.6 and A.2.7

$$X \leq_{st} \sum_{i=1}^{\alpha-1} X_i + 1 \leq_{st} \sum_{i=1}^{\alpha-1} Y_i + 1 \leq_{st} B(n^2, \frac{2d}{n^2}) + 1 \leq_{st} Y + 1 \leq_{st} Z.$$

□

אוניברסיטת תל-אביב TEL AVIV UNIVERSITY

הפקולטה למדעים מדויקים ע"ש ריימונד ובברלי סאקלר

בית הספר למדעי המחשב ע"ש בלבטניק

תכנון אלגוריתמי ומנגנוני חישוב מקומי

חיבור לשם קבלת תואר "דוקטור לפילוסופיה"

מאת

שי ורדי

בהנחייתו של

פרופסור ישי מנצור

הוגש לסנאט של אוניברסיטת תל אביב

ספטמבר 2015

תמצית

אלגוריתמי חישוב מקומיים (אח״מים) מאפשרים גישה לחלקים קטנים של פתרון לבעיה נתונה, תוך כדי שימוש בזמן ומקום פולילוגאריתמיים באורך הקלט. הניחו, למשל, שיש גרף ענק עליו אנו רוצים לחשב קבוצה בלתי תלויה מקסימאלית של קודקודי הגרף. הגרף כל כך גדול שחישוב כל הפתרון ייקח יותר מדי זמן, ובנוסף, יתכן ואין לנו מספיק מקום כדי לאחסן את הפתרון כולו. מצד שני, נניח שבכל זמן נתון, איננו צריכים את הפתרון כולו. במקום זאת, מדי פעם אנו נשאלים לגבי קודקודים מסוימים – האם הם נמצאים בקבוצה הבלתי-תלויה. אח״מ יאפשר לנו להגיב לשאילתות אלה באופן עקבי (כלומר, אם האח״מ נשאל לגבי כל קודקודי הגרף, כל התשובות תהיינה עקביות עם פתרון יחיד), תוך שימוש בזמן ובמרחב פולילוגאריתמיים. בתזה זו אנו מפתחים טכניקות וכלים לתכנון וניתוח אח״מים.

אנו מציגים משפחה חדשה של גרפים – $\mathrm{d}-\mathrm{light}$, הכוללת גרפים בעלי דרגה חסומה ומספר גרפים אקראיים מעניינים. אנו מראים כיצד ניתן לעשות רדוקציה מאח״מים לבעיות מסוימות על גרפי $\mathrm{d}-\mathrm{light}$ לאלגוריתמים מבוזרים ומקוונים. הרדוקציה לאלגוריתמים מבוזרים הוצגה ע״י Parnas and Ron (2007). הניתוח שלה על גרפים $\mathrm{d}-\mathrm{light}$ חדש. אנו מציגים רדוקציה לאלגוריתמים מקוונים המבוססת על רעיון של Nguyen and Onak (2008): הרעיון הוא לייצר סדר אקראי על הקודקודים ולדמות הרצה של אלגוריתם מקוון על סדר זה. אנו מראים כי מספיק לשמור גרעין של אקראיות באורך $O(\log n)$ בכדי לייצר את האקראיות שאנו צריכים, ושבהסתברות גבוהה האח״מים שנוצרים לא משתמשים ביותר מ $O(\log^2 n)$ זמן או מקום להשיב לכל שאילתא. רדוקציות אלו מאפשרות לנו להשיג אח״מים לבעיות כגון: קבוצה בלתי תלויה מקסימאלית, זיווג מקסימלי, איזון עומסים וצביעה על גרפים $\mathrm{d}-\mathrm{light}$. אנו מרחיבים את הטכניקות הללו כדי להשיג אח״מ קירוב לזיווג מקסימלי על גרפים בעלי דרגה קבועה. בנוסף, אנו מראים כי במקרים מסוימים, אנו יכולים לעצב אח״מים עבורם זמן הריצה (והמקום) אינו תלוי בגודל הגרף, אלא רק בדרגה המקסימלית. ליתר דיוק, אנו מציגים אח״מ קירוב לבסיס משוקלל מקסימאלי של מטרואיד גרפי (כלומר, קבוצת קשתות נטולת מעגלים מקסימאלית), אח״מים לקירוב חתך מרובה וזרימה מרובת-משאבים על עצים, ורדוקציה מקומית של זיווג ממושקל מקסימאלי לאח״מ לזיווג מקסימאלי, כך שזמן ריצתו של האח״מ המיוצר בלתי תלויה במשקלי הקשתות.

תחום עיצוב מנגנונים (mechanism design) כרוך בתכנון אלגוריתמים לסביבות אסטרטגיות - כאלה שהקלט לאלגוריתם מורכב (או כולל) מידע פרטי שנמסר ע״י סוכנים שיש להם עניין בתוצאה. אנו מציגים את התחום ״עיצוב מנגנון חישוב מקומי״, הכולל עיצוב מנגנונים המקבלים שאילתא על סוכן יחיד, ונדרשים להחזיר

את החלק מהפתרון שרלוונטי לסוכן, תוך שמירה על המאפיינים הנדרשים מהמנגנון. כדוגמא, נניח ויש מכירה פומבית של מיליון פריטים למיליוני סוכנים. כאשר אנו מתשאלים את המנגנון לגבי סוכן, היינו רוצים שהוא יגיד לנו אילו פריטים הסוכן מקבל ומה התשלום שהוא חייב עבור פריטים אלו. בו בעת היינו רוצים לתמרץ את כל הסוכנים לדווח בצורה כנה על הערכותיהם לשווי הפריטים.

בדומה לאח״מים, מנגנוני חישוב מקומיים נדרשים להגיב לכל שאילתא בזמן ובמרחב פולילוגאריתמיים, ועל התגובות לשאילתות לעלות בקנה אחד עם אותו פתרון גלובלי. כאשר המנגנון משתמש בתשלומים, חישוב התשלומים נעשה גם הוא בזמן ובמרחב פולילוגאריתמיים. יתר על כן, על המנגנון לתמרץ את המשתתפים לדווח בצורה כנה.

אנו מציגים מנגנוני חישוב מקומיים עבור מגוון רחב של בעיות בתורת המשחקים: (1) זיווג יציב, (2) איזון עומסים, (3) מגוון מכירות פומביות קומבינטוריות ו- (4) הקצאת דיור. לחלק מהטכניקות שלנו יש השלכות לבעיית זיווג יציב הכללית (לא מקומית): אנו מראים כי כאשר רשימות העדפה של הגברים חסומות, אנחנו יכולים להשיג זיווג המהווה קירוב טוב לזיווג היציב תוך מספר קבוע של סיבובים של אלגוריתם Gale-Shapley.

תקציר

חלק 1 : תכנון אלגוריתמי חישוב מקומי

בחלק הראשון של התזה, אנו מציגים את המודל של אלגוריתמי חישוב מקומיים (אח״מים) ומתארים מספר טכניקות לעיצוב אח״מים. בפרק 2 אנחנו מציגים את המודל ומגדירים את חמשת הקריטריונים לביצועים של אח״מים על גרפים : מספר הגישות שהאח״מ מבצע לגרף עבור כל שאילתא ; הזיכרון הנדרש כדי לאחסן את גרעין האקראיות, (אם האח״מ אקראי) ; הזמן הנדרש לענות לכל שאילתא ; המקום הנדרש לחישוב התשובה לשאילתא ; וההסתברות שהאח״מ חורג ממספר הגישות, הזמן או המקום.

לאחר מכן, אנו מציגים משפחה של גרפים, אותה אנו מכנים $d-\text{light}$: גרף $d-\text{light}$ הוא גרף שנדגם מהתפלגות, כך שכאשר תת-גרף כלשהו כבר נחשף, דרגת הקודקוד הנחשף הבא נשלטת ע״י התפלגות קלת-זנב בעלת תוחלת d. משפחה זו כוללת מספר רב של גרפים, כולל גרפים בעלי דרגה חסומה וגרפים אקראיים. אנו מראים כי בהינתן שתת-גרף גדול מספיק נחשף, השכונה של תת-הגרף לא הרבה יותר גדולה מתת-הגרף עצמו.

בסופו של פרק 2, אנו מראים כי קיימות בעיות, אפילו כאלה שניתן לפתור בזמן פולינומי, שאין להן אח״מים. ליתר דיוק, אנו מראים כי לא קיים אח״מ לזיווג מקסימום.

בפרקים 3 ו-4 אנו מראים כיצד ניתן להמיר אלגוריתמים מבוזרים ומקוונים מסוימים, בהתאמה, לאח״מים. הרעיון מאחורי הרדוקציה לאלגוריתמים מבוזרים הוא כדלקמן : אם אנו נשאלים לגבי קודקוד מסוים, v, ויש לבעיה אלגוריתם מבוזר D המסיים את ריצתו לאחר מספר קבוע, k, של סיבובים, ניתן לדמות את הרצת D על כל הקודקודים במרחק לכל היותר k מ-v. אם הגרף הוא $d-\text{light}$, אנו מראים כי מספר הקודקודים האלה הוא בהסתברות גבוהה לוגריתמי בגודל הגרף. הרדוקציה לאלגוריתמים מקוונים היא כדלקמן : אנו מייצרים סדר אקראי על הקודקודים, ומדמים אלגוריתם מקוון A על סדר זה. מכיוון שאנו רוצים שהתשובות לכל השאילתות תהיינה עקביות עם פתרון יחיד, עלינו לדמות את A על אותו הסדר עבור כל השאילתות. לכן עלינו לשמור את הסדר הזה. למרבה הצער, התוצאות שלנו תלויות בכך שהסדר אקראי, ולכן איננו יכולים להשתמש בסדר שרירותי (למשל, לפי מספר סידורי). למרות שאנו דורשים שהסדר יהיה אקראי, אנחנו לא דורשים אקראיות מוחלטת. באופן כללי, ככל שיש לנו יותר מקום, אנו יכולים לקבל יותר אקראיות. מתברר כי מספר לוגריתמי של ביטים אקראיים מספיק למטרותינו. מגרעין קטן זה, אנו יכולים ליצור סדר ״כמעט״ אקראי על הקודקודים, שעדיין מבטיח שנצטרך לגשת לגרף מספר פולילוגאריתמי של פעמים עבור כל

שאילתא במקרה הגרוע ביותר, בהסתברות גבוהה. אנו גם מראים כי בתוחלת, אנו צריכים מספר קבוע של גישות לגרף עבור כל שאילתא. בפרק 5 אנו מראים אח״מ לקירוב זיווג מקסימום על גרפים בעלי דרגה חסומה.

בסופו של החלק הראשון של התזה אנו מראים כי לעיתים נוכל לתכנן אח״מים שזמן ריצתם אינו תלוי בגודל הגרף, אלא רק בדרגה המקסימאלית (פרק 6). אנו מציגים אח״מ למציאת יער שמשקלו קירוב $1-\varepsilon$ למשקל היער פורש המקסימאלי, ואח״מים לחישוב זרימה רבת-משאבים אינטגראלית וחתכים מרובים על עצים שמהווים 4- ו- 1/4- קירוב לפתרונות האופטימאליים בהתאמה. לבסוף, אנו מראים כי, בהינתן אלגוריתם שנותן קירוב α לזיווג מקסימום (לא ממושקל) שרץ בזמן קבוע, ניתן לקבל אלגוריתם המשיג קירוב $\alpha/8$ לזיווג מקסימום ממושקל. זמן הריצה של האח״מ אינו תלוי בפונקצית המשקל או בגודל הגרף.

חלק 2 : תכנון מנגנוני חישוב מקומי

בחלקו השני של התזה, אנו מתכננים מנגנוני חישוב מקומי - מנגנונים (במובן של תורת המשחקים), הדורשים זמן ומקום פולילוגאריתמי. בפרק 7 אנו מגדירים את המודל ונותנים דוגמא פשוטה - יישום אלגוריתם דיקטטורה סידורית אקראית. בבעיית הקצאת הדיור, יש קבוצת בתים וקבוצת קונים, ויש לכל קונה רשימת עדיפות על פני הבתים. ברצוננו למצוא הקצאה ״טובה״ של בתים לקונים. יש לאלגוריתם הדיקטטורה הסידורי האקראי כמה תכונות רצויות, כגון היעדר תמריץ לקונים לשקר לגבי רשימת העדפותיהם. המנגנון המקומי דורש שהקלט יקיים מגבלות מסוימות (בעיקר שהרשימה של כל קונה היא באורך קבוע, והבתים בה נבחרים אקראית באופן אחיד), וכאשר הוא נשאל על קונה, האח״מ מחזיר את הבית שמוקצה לו, תוך שמירה על התכונות הרצויות של המנגנון הלא מקומי.

בפרק 8, אנו מראים כיצד ליישם את אלגוריתם הזיווג היציב של Gale-Shapley בתור אח״מ כאשר רשימת כל גבר מכילה מספר קבוע של נשים, שנבחרו אקראית באופן אחיד. באלגוריתם Gale-Shapley, כל גבר ניגש לאישה המועדפת ביותר עליו. האישה מקבלת (טנטטיבית) את הגבר שהיא מעדיפה מכל הגברים שניגשו אליה, ודוחה את כל השאר. בסיבוב הבא, כל גבר שדחה בסיבוב הקודם ניגש לאישה הבאה ברשימה שלו. כל אישה שוב מקבלת את הגבר שהיא מעדיפה (מתוך כל הגברים שעומדים מולה, כולל זה שהיא קיבלה בסיבוב הקודם, אם היה כזה). תהליך זה ממשיך עד שכל גבר עומד מול אישה אחת. גייל ושפלי הראו כי זיווג זה יציב. היישום המקומי שלנו פשוט : אנו מדמים את אלגוריתם Gale-Shapley למספר קבוע, k, של סיבובים, כאשר כל גבר שנדחה בסיבוב ה- k נפסל. אנחנו מראים שעל ידי בחירת מספר הסיבובים, אנו יכולים להבטיח כי חלק קטן כרצוננו מהגברים נפסל. זה מבטיח כי הזיווג שלנו הוא כמעט יציב (ז״א קיימים מעט מאוד זוגות של

גברים ונשים שמעדיפים אחד את השני על פני בן הזוג שלהם בזיווג.) אנו משתמשים בטכניקות שפיתחנו להראות, שבמקרה הכללי, כאשר הרשימות של הגברים באורך חסום (לא רק כאשר מדובר באח״מים), אנו יכולים למצוא זיווג כמעט יציב טוב כרצוננו על ידי הרצת אלגוריתם Gale-Shapley למספר קבוע של סיבובים.

בפרק 9 אנו חוקרים את הבעיה של תזמון עבודות על מכונות: אנחנו רוצים לתזמן n עבודות על m מכונות כדי למזער את זמן הריצה המקסימלי של המכונות. יש לבעיה זו וריאציות רבות. אנו בוחנים את התרחיש שבו עלינו לתזמן n עבודות זהות על m מכונות קשורות. המכונות הן סוכנות אסטרטגיות, עבורן המידע הפרטי הוא מהירותן. אנחנו מראים:

1. מנגנון מקומי כן בתוחלת לתזמון על מכונות קשורות, הנותן קירוב לוגלוגאריתמי לפתרון האופטימלי.
2. מנגנון מקומי כן בצורה אוניברסלית לתזמון על מכונות מוגבלות (כלומר, כאשר כל עבודה יכולה להתבצע על לכל היותר מספר קבוע של מכונות שנקבעו מראש), הנותן קירוב לוגלוגאריתמי לפתרון האופטימלי.

כמו כן, אנו מראים כמה תוצאות עדינות ומפתיעות על הכנות של האלגוריתמים שלנו.

בפרק 10, אנו חוקרים מכרזים קומבינטורים הקשורים לזיווגים. מכרזים קומבינטורים הם מכרזים בהם קונים יכולים להציע מחיר על חבילות של פריטים. אנו בוחנים את התרחיש הבא: m קונים משתתפים במכרז עבור n פריטים, כאשר כל קונה מעוניין בקבוצה של לכל היותר k פריטים, הנדגמים אקראית. כמו כן, כל קונה מעוניין לקבל לכל היותר פריט אחד מהקבוצה (לא משנה איזה). אנחנו מראים מנגנונים מקומיים כנים בצורה אוניברסלית לווריאציות הבאות.

שני המנגנונים נותנים קירוב ½ לפתרון האופטימאלי (מבחינת רווחה חברתית).

1. כשיש לכל הקונים את אותה הערכת שווי לפריטים בהם הם מעוניינים, והמידע הפרטי של הקונים הוא קבוצות הפריטים בה הם מעוניינים.
2. כשהקבוצות ידועות, והמידע הפרטי של הקונים הוא הערכת השווי שלהם עבור הפריטים בהם הם מעוניינים (תחת ההנחה שהשווי זהה עבור כל הפריטים).

אם כל קונה מעוניין בקבוצת פריטים בגודל לכל היותר k, ויש לו הערכת שווי פרטית עבור קבוצה זו, אנו מראים שקיים מנגנון מקומי כן בצורה אוניברסלית המוצא פתרון שמהווה קירוב $1/k$ לפתרון האופטימלי ביחס לרווחה החברתית.