Graph Algorithms in the Internet Age

by

Isabelle Lesley Stanton

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Satish Rao, Chair
Professor Elchanan Mossel
Professor Dorit Hochbaum

Fall 2012

Graph Algorithms in the Internet Age

Abstract

Graph Algorithms in the Internet Age

by

Isabelle Lesley Stanton

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Satish Rao, Chair

This dissertation addresses a series of graph problems inspired by the computational issues with face with the Internet, a massive distributed network of autonomous agents. There are several levels to this problem. From a systems perspective, what can we do to facilitate computation over massive graphs? From a modeling perspective, what do natural graphs look like and what features are useful? From a game theoretic perspective, the graphs often represent individuals or systems with their own goals and agendas. Can we understand how these systems compete and when these competitions are fair or can be manipulated?

These questions are addressed. For the first, we consider the problem of streaming graph partitioning and show it is feasible. For the second, we study the joint degree distribution of a graph and show it is combinatorially easy to work with. Finally, we address questions about tournament design and manipulation.

For Dad, who struggled for so long and made great sacrifices for all of his childrens' education. He would have been 'insufferably' proud to see this document.

Contents

List of Figures

List of Tables

Acknowledgments

I have received significant technical advice and assistance in all of this work, first and foremost from my advisor, Satish Rao, and from Virginia Vassilevska Williams. Beyond that, I have been fortunate that many people have been willing to discuss my projects with me. In particular, for each chapter, I would like to acknowledge the input of:

Chapter 3 Gabriel Kliot first brought this problem to my attention and assisted greatly in developing a model for the experiments that matched reality. In addition, I'd like to acknowledge the many conversations with Microsoft researchers that inspired this work, in particular those with Nikhil Devanur, Sameh Elkinety, Sreenivas Gollapudi, Yuxiong He, Nina Mishra, Rina Panigrahy, Yuval Peres, Burton Smith and David B. Wecker. The paper would never have been published if Matei Zaharia had not selflessly run my Spark experiments.

Chapter 4 This chapter is the direct outcome of Satish pestering me about why the greedy algorithms worked (and behaved so differently) in the previous chapter. Conversations with Miklos Racz, Alexandre Stauffer and Ngoc Mai Tran helped me work out the exact details of the coupling argument.

Chapter 6 I am deeply grateful to Alistair Sinclair for directing me towards autocorrelation as a method of evaluation mixing time. Also, David Gleich was a great help in finding related work.

Chapters 8 and 9 I was introduced to the problem of fixing tournaments by Virginia who patiently spent many hours finding and fixing the holes in my proofs.

Berkeley and the Theory group I have had a rich and satisfying experience as a graduate student at Berkeley, in large part due to the diversity and strength of the students, particularly those in the RadLab, AmpLab and ParLab. The Theory group itself is extremely diverse and the exposure to all of these ideas has been wonderful. Thanks to my office mates, Raf and Anindya, for answering all my questions whenever they dared enter the office. Thanks to Yaron for being a good first year mentor and pointing out that Theory really can matter in practice. Thanks to Greg for making sure that there was sufficient coffee available to power the rest of the group. Thanks to Alex for teaching me how to surf and rock climb and reminding me there is more to life than writing research papers.

Funding. I have been fortunate to have ample funding for my graduate career including the NSF Graduate Fellowship, NDSEG Graduate Fellowship, and NPSC Graduate Fellowship. Additional funding came from scholarships including the Google Anita Borg and Yahoo!

Key Scientific Challenges, NSF grants CCF-0830797 and CCF-1118083, and an uncountable number of travel grants.

Finally, I'd like to thank both of my parents, not for encouraging me, but for never having a doubt that completing this degree was something I could, would, and should do.

Chapter 1

Introduction

Arguably, one of the most culturally interesting artifacts of computing is the Internet. Its existence and spread has had a vast impact on society globally, from the advant of online social networking and smart phones to relatively more important historical events like the Arab Spring. Its ubiquity has had a dramatic impact on all areas of Computer Science, from networking, to the entire paradigm shift in Systems to cloud computing, all the way through to Theoretical Computer Science where we are beginning to understand and rediscover the importance of streaming, distributed and parallel algorithms.

The advances in storage technology, combined with the networked nature of the applications, have generated a vast array of massive, heterogeneous datasets (an interesting problem for the Databases community). These new applications emphasize the importance of a series of interesting algorithmic questions:

- How can we effectively design distributed or parallel algorithms?
- Given the massive size of the data, sometimes ranging into Petabyes, can we design algorithms to compute solutions without the traditional random access assumption?
- The sheer size of the data can make computing a solution a daunting task. When this data changes slightly, how can we effectively reuse the previous computation time to find the new answer? What if the data is being changed adversarially?
- Each part of the network is an autonomous agent, with its own goals and desires. How can we employ game theory to design algorithms that align the incentives of these agents so that they work together towards the same goal?
- Often, as we develop algorithms for large data problems, we would like to evaluate their performance. Waiting for them to run on the full data set can take a prohibitively long time to be used in a test and debug cycle. Can we develop models to generate synthetic data sets with the same important features?

- Graph data is notoriously difficult to optimize, due to its quadratic size and non-linear nature. Given the massive amount of graph data we are collecting, what can we do to ease this problem?
- How does the above question change when we change the graph type from the traditional scientific computing finite element mesh to the friend graph of a social network?

Perhaps the most important question is whether we can use existing algorithmic tools to solve these problems. This thesis will primarily focus on answering a selection of the above questions as applied to graph algorithm problems.

Part I The first part of this thesis focuses on the question of streaming graph partitioning. Graph partitioning has been studied by theoretical computer scientists for decades and its variants form one of the most fundamental algorithm questions. The most general way to describe all of these variants is that they concern themselves with cutting a graph into two or more pieces while minimizing the number of edges cut between the pieces. With no additional constraints, this is the classic MIN-CUT problem, solved in 1954 by Ford and Fulkerson's maximum flow algorithm [57].

The many variants come from the vast array of applications. For example, MAX-FLOW/MIN-CUT was studied because it modeled a military application - what are the weak links in an enemy's rail system so that you can efficiently cut off their supply lines? The variant I consider arises from considering the communication patterns of a distributed computation. If we must split this computation over many machines, each should have an equal load and we would like to minimize the communication needed between each machine.

This problem is known as k-balanced partitioning and has nearly as long of a history, both in the Theory community and Scientific Computing community, as all of graph partitioning. In this work, we consider what happens when the communication graph is too large to fit in one machine's memory. This becomes a catch-22 - in order to efficiently partition the data, we must first partition the communication graph of the partitioning algorithm. This new graph may closely match the original graph in size.

We instead consider a streaming approach. Rather than assuming we can fit the entire graph into main memory, we attempt to partition it vertex by vertex as it arrives from a generating source, perhaps a web crawler or perhaps from a file on disk. The first chapter focuses on experimentally evaluating various algorithms for this problem with the goal of demonstrating that good algorithms for this problem do exist. The chapter explores sixteen different streaming partitioning heuristics and evaluates their performance on twenty-two graphs of varying sizes. The second chapter provides a theoretical analysis of the best performing algorithm and a very closely related variant. This analysis is completed by coupling the algorithms to finite Polya Urn processes. The proof clearly demonstrates the differences in performance observed in the first chapter between the two variants.

Part II The second part of the thesis rather than focusing on a set of related problems, focuses on using a specific technique, finding a matching in a graph, to solve a variety of problems.

The first problem is to study a graph feature, the joint degree distribution. One of the most fundamental results in the study of social and information networks is that the degree distribution, a histogram of how many people have 1, 5, 100, or 1000 friends, follows a power law distribution where the fraction of the people with k friends is proportional to $k^{-\alpha}$ for some α. This result is fundamentally surprising because it means this quantity in these networks does not follow the Central Limit Theorem and become normal. The joint degree distribution measures not how many friends a person has but the distribution over the edges - does this network tend to have low degree nodes connecting to other low degree nodes or high degree nodes? Many existing models of social and information networks fail to adequately match reality, and one reason may be that the joint degree distribution can be wildly different for different types of graphs, yet all are supposed to fit into the same model. In Chapter 4, I address the feasibility of using the joint degree distribution within these models. Given a joint degree distribution, can we decide if it matches a real graph, or do some of the constraints conflict? If it does match, can we reconstruct this graph? Finally, if we can reconstruct it, can we sample a graph uniformly at random that does? This question about uniform sampling allows us to measure which other features are correlated with this one, as well as run unbiased experiments.

The second problem appeals to the problem of understanding how selfish agents act when asked to participate in a protocol, in particular, a voting or election protocol. Gibbard and Satterthwaite [61, 128] independently showed that any reasonable social choice function is susceptible to *tactical voting* where an individual voter can change their vote to obtain a better result for themselves. The existence of tactical voting is fundamentally troubling from two standpoints. The first is that the ideal goal of a social choice function is to produce an outcome that is optimal in terms of social welfare. Tactical voting undermines this goal by forcing less optimal solutions be selected. The second is that the existence of tactical voting makes the mechanism extremely difficult to use 'If I know that you will vote this way, then I should vote that way. However, you also know that I will vote that way, so then you will vote a third way.' and no resolution can be reached. Fortunately, Bartholdi, Tovey and Trick [25] introduced the idea of using computational complexity to sidestep tactical voting. If, given knowledge of everyone else's votes, it is computationally hard for me to compute if there exists a better vote for me then I will just vote with my true preferences.

This observation has spawned an entire field of study, computational social choice. While much of the work has focused on proving that existing voting rules are NP-hard, this chapter focuses on the *binary cup* voting rule (or single-elimination tournaments). Some hardness results are known for variants, but we look at a realistic case and study how much power the election organized has over manipulating the outcome of the mechanism. We show that for candidates satisfying reasonable conditions the organizer can find a manipulation very quickly. We view this result as showing that average case hardness is the ideal measure for computational social choice, as NP-hardness is not enough to protect voters in most

reasonable settings.

The final chapter was inspired by trying to answer the above manipulation questions about double-elimination tournaments instead of single-elimination. Despite the widespread use of double-elimination tournaments in a variety of applications from, obviously, competitions, to, less obviously, medical experiment design and hearing aid fitting, there is no existing definition of their structure in the literature. To address this, we consider several simple tournament design goals - that they should be fair, balanced, and avoid unnecessary repeated matches - and formalize what each of goals means in a mathematical sense. While there is no single definition, there is a similar family of double-elimination structures used in practice, and we show that these are not optimal with respect to avoiding repeats. We suggest a minor change that does make them optimal, and show that this change does not negatively impact the tournaments ability to select strong players as the winner.

Part I

New Approaches to Graph Partitioning Problems

Chapter 2

An Introduction to Streaming Graph Partitioning

Modern graph datasets are huge. The clearest example is the World Wide Web where crawls by large search engines currently consist of over one trillion links and are expected to exceed ten trillion within the year. Individual websites also contain enormous graph data. In Jan 2012, Facebook consisted of over 800 million active users, with hundreds of billions friend links [2]. There are over 900 million additional objects (communities, pages, events, etc.) that interact with the user nodes. In July 2009, Twitter had over 41.7 million users with over 1.47 billion social relations [83]. Since then, it has been estimated that Twitter has grown to over 200 million users. Examples of large graph datasets are not limited to the Internet and social networks - biological networks, like protein interaction networks, are of a similar size. Despite the size of these graphs, it is still necessary to perform computations over the data, such as calculating PageRank, broadcasting Twitter updates, identify protein associations [21], as well as many other applications.

The graphs consist of terabytes of compressed data when stored on disks and are all far too large for a single commodity type machine to efficiently perform computations. A standard solution is to split the data across a large cluster of commodity machines and use parallel, distributed algorithms for the computation. This approach introduces a host of systems engineering problems of which we focus only on the problem of data layout. For graph data, this is called *balanced graph partitioning*. The goal is to minimize the number of cross partition edges, while keeping the number of nodes (or edges) in every partition approximately even.

Good graph partitioning algorithms are very useful for many reasons. First, graphs that we encounter and care about in practice are not random. The edges display a great deal of locality, whether due to the vertices being geographically close in social networks, or related by topic or domain on the web. This locality gives us hope that good partitions, or at least partitions that are significantly better than random cuts, exist in real graphs. Next, inter-machine communication, even on the same local network, is substantially more expensive than inter-processor communication. Network latency is measured in microseconds while

inter-process communication is measured in nanoseconds. This disparity substantially slows down processing when the network must be used. For large graphs, the data to be moved may border on gigabytes, causing network links to become saturated.

The primary problem with partitioning complicated graph data is that it is difficult to create a linear ordering of the data that maintains locality of the edges i.e. if it is possible to embed the vertices of a graph into a line such that none of the edges are 'too long', then a good balanced cut exists in the graph. Such an ordering may not even exist at all. There is a strong connection between graph partitioning and the eigenvectors and eigenvalues of the corresponding Laplacian matrix of the graph via the Cheeger bound. This connection has inspired many spectral solutions to the problem, including ARV [18] and the many works that followed.

However, spectral methods do not scale to big data. This is in part due to the running time and in part because current formulations require full graph information. When a graph does not physically fit on one machine, maintaining a coherent view of the entire state is impossible. This has led to local spectral partitioning methods, like EvoCut [15], but local methods still require access to large portions of the graph, rely on complex distributed coordination and large computation after the data has been loaded. Thus, we look for a new type of solution. A *graph loader* is a program that reads serial graph data from a disk onto a cluster. It must make a decision about the location of each node as it is loaded. The goal is to find a close to optimal balanced partitioning with as little computation as possible. This problem is also called *streaming graph partitioning*.

For some graphs, partitioning can be entirely bypassed by using meta data associated with the vertices, e.g. clustering web pages by URL produces a good partitioning for the web. In social networks, people tend to be friends with people who are geographically nearby. When such data is available, this produces an improved cut over a node ID hashing approach. Unfortunately, this data is not always be available, and even if it is, it is not always clear which features are useful for partitioning. Our goal in this work is to find a general streaming algorithm that relies only on the graph structure and works regardless of the meta data.

2.1 Applications

Our motivating example for studying this problem is a large distributed graph computation system. All distributed computation frameworks, like MapReduce, Hadoop, Orleans [34] and Spark [153] have methods for handling the distribution of data across the cluster. Unfortunately, for graphs, these methods are not tuned to minimize communication complexity, and saturating the network becomes a significant barrier to scaling the system up.

The interest in building distributed systems for graph computation has recently exploded, especially within the database community. Examples of these systems include Pregel [99], GraphLab [94], InfiniteGraph, HyperGraphDB, Ne04j, and Microsoft's Trinity [4] and Horton [3], to name but a few. Even for these graph specific systems, the graphs are laid out using a hash of the node ID to select a partition. If a good pseudorandom hash function

is chosen, this is equivalent to using a random cut as graph partitioning and will result in approximately balanced partitions. However, computations on the graph run more slowly when a hash partitioning is used instead of a better partitioning, due to the high communication cost. Fortunately, these systems tend to support custom partitioning, so it is relatively easy to substitute a more sophisticated method, provided it scales to the size of the graph. As our experiments show, even using our simple streaming partitioning techniques can allow systems of this type to complete computations at least 20-40% faster.

2.2 Theoretical Difficulties - Lower Bounds

Theoretically, a good streaming partitioning algorithm is impossible. It is easy to create graphs and orderings for any algorithm that will cause it to perform poorly. A simple example of this lower bound is a cycle. The optimal balanced 2-partition cuts only 2 edges. However, if the vertices are given in an order of 'all even nodes then all odd nodes', we won't observe any edges until the odd nodes arrive. Without any information, the best an algorithm could is to try and balance the number of vertices it has seen across the 2 partitions. This leads to an expected cut of $\frac{n}{4}$ edges. The worst algorithm might put all even nodes in one partition leading to all edges being cut!

We can partially bypass this problem by picking the input ordering. The three popular orderings considered in the literature are adversarial, random, and stochastic. The above cycle example is an adversarial order and demonstrates that the streaming graph partitioning problem may have arbitrarily bad solutions under that input model. Given that adversarial input is unrealistic in our setting - we have control over the data - we focus on input that results from either a random ordering, or the output of a graph search algorithm. The second option is a simplification of the ordering returned by a graph crawler.

Theorem 1. *One-pass streaming balanced graph partitioning with an adversarial stream order can not be approximated within $o(n)$.*

Proof. Without loss of generality, we seek a balanced 2 partitioning. Consider a graph that is a cycle over n vertices with edges such that $(i, i+1) \mod n \in E$ for $1 \leq i \leq n$. Let the ordering be all odd nodes, then all even, i.e. $1, 3, 5 \ldots n-1, 2, 4, 6 \ldots n$. Assume that n is even. The optimal balanced partitioning cuts 2 edges. However, the given ordering reveals no edges until $\frac{n}{2}$ vertices arrive. Until the edges arrive, we have no way of distinguishing which vertices are 'near' each other. In particular, note that this ordering is indistinguishable from one where the odd vertices are given in a random order, or one where the odd nodes are interspersed with unconnected even nodes, i.e. $1, n-2, 3, n-4, 5, n-6 \ldots$. Thus, no algorithm can do better than cutting $\frac{n}{2}$ edges in expectation. This generalizes to k partitions. □

Theorem 2. *One-pass streaming balanced graph partitioning with a random stream order can not be approximated within $o(n)$.*

Proof. Again, we seek a balanced 2 partition for a cycle graph with a random ordering. Consider the t^{th} vertex to arrive in this ordering.

$$\Pr[t \text{ arrives with no edges }] = \Pr[\text{ both neighbors arrive after } t] = \frac{n-t}{n}\frac{n-t-1}{n-1}$$

so the number of vertices that we expect to arrive with no edges is

$$\mathbf{E}[\#\text{ with no edge}] = \sum_{t=1}^{n} \frac{t}{n}\frac{t+1}{n-1}$$

$$\approx \frac{1}{n^2}\sum_{t=1}^{n} t^2 - t = \frac{1}{n^2}(\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} + \frac{n(n+1)}{2})$$

Therefore, asymptotically, we expect $\frac{n}{3}$ vertices to arrive with no edges. As before, when a vertex arrives with no edges, we are not able to determine which other vertices it is 'near'. For each of these, we can expect to cut 1 edge, providing us with our lower bound. □

We leave open the problem of providing lower bounds for other stream orderings. While there are current approaches to analyzing streaming algorithms, our use of breadth-first and depth-first stream orders is novel and previous approaches can not be applied. Theorem 5 shows we should not hope to analyze any algorithm with an adversarial order, while a random ordering will always hide edges in sparse graphs until $O(\sqrt{n})$ vertices arrive, making competitive analysis difficult.

2.3 The Streaming Model

We consider a simple streaming graph model. We have a cluster of k machines, each with memory capacity C, such that the total capacity, kC, is large enough to hold the whole graph. The graph is $G = (V, E)$ where V is the vertices, and E the edges. The graph may be either directed or undirected. The vertices arrive in a stream with the set of edges where it is a member so for undirected graphs, each edge appears twice in the stream. We consider three orders: random, breadth-first search and depth-first search. As vertices arrive, a partitioner decides to place the vertex on one of the k machines. A vertex is never moved after it has been placed. In order to give the heuristics maximal flexibility, we allow the partitioning algorithm access to the entire subgraph defined by all vertices previously seen. This is a strong assumption, but the heuristics studied in this thesis use only local (depth 1) information about this subgraph. We extend the model by allowing a buffer of size C so that the partitioning algorithm may decide to place any node in the buffer, rather than the one at the front of the stream.

Our model assumes serial input and a single loader. This is somewhat unrealistic for a real system where there may be many graph loaders working in parallel on independent portions

of the stream. While we will not explore this option, the heuristics we investigate can be easily adapted to a parallel setting where each loads its portion of the graph independently from the others, sharing information only through a distributed lookup table of vertices to partition IDs.

2.4 Related Work

Graph partitioning has a rich history. It encompasses many problems and has many proposed solutions, from the very simple to the very sophisticated. We cannot hope to cover the whole field and will only focus on the most relevant formulation - balanced k-partitioning. The goal is, given a graph G as input and a number k, to cut G into k balanced pieces while minimizing the number of edges cut. This problem is known to be NP-Hard, even if one relaxes the balanced constraint to 'approximately' balanced [16]. Andreev and Racke give an LP-based solution that obtains a $O(\log n)$ approximation [16]. Even *et al.* [54] provide another LP formulation based on spreading metrics that also obtains an $O(\log n)$ approximation. f one ignores the balance constraint, a popular approach is to use the top k eigenvectors [115]. Recently, this approach was theoretically validated as an extension of Cheeger's inequality [88, 93]. Both require full information about the graph.

There are many heuristics that solve this problem with an unknown performance guarantee, like METIS [77], PMRSB [23], and Chaco [70]. In practice, these heuristics are quite effective, but many are intended for scientific computing purposes. One can recursively use any balanced 2-partitioning algorithm to approximate a balanced k-partitioning when k is a power of 2 [18].

Another approach, relevant for our limited information setting, is *local partitioning* algorithms. The goal here is not to obtain a balanced cut but given a starting node to find a good cut around that node. Spielman and Teng were the first to develop this style of algorithm [135]. Anderson, Chung and Lang improved upon Spielman and Teng's work by using personalized PageRank vectors to find a good local cut [14]. Addressing the same problem, Anderson and Peres use the evolving graph process to obtain similar results [15]. While local partitioning is similar in spirit, using local partitioning to find a balanced cut is unsolved.

While we are unaware of any previous work on the exact problem statement that we study - one pass balanced k partitioning - there has been much work on many related streaming problems, primarily graph sparsification in the semi-streaming model, cut projections in the streaming model as well as online algorithms in general, like online bipartite matching. This work includes both algorithms and lower bounds on space requirements.

The work on streaming graph problems where multiple passes on the stream are allowed includes estimating PageRank [127] and cut projections [126]. While PageRank has been used for local partitioning [14], the approach in [14] uses personalized PageRank vectors which does not easily generalize the approach in [127]. Additionally, cut projections do not maintain our balanced criterion.

The main focus of the next few chapters is on streaming algorithms and there is significant related work in this area as well. First, noting the connection between graph partitioning and PageRank is Das Sarma et al.'s work on computing the PageRank of a graph with multiple passes [127]. Closer to our setting, Bahmani et al. incrementally compute an approximation of the PageRank vector with only one pass [20]. However, just computing the final PageRank vector is not sufficient for finding a graph partitioning. Das Sarma et al. extend their techniques to find sparse cut projections within subgraphs, again using multiple passes over the stream [126]. Cut projections are not the same as finding balanced cuts.

An alternate model, *semi-streaming*, assumes that all vertices are known from the beginning but the edges arrive in an adversarial order. In this setting, Ahn and Guha [6] give a one pass $\tilde{O}(n/\epsilon^2)$ space algorithm that sparsifies a graph such that each cut is approximated to within a $(1+\epsilon)$ factor. Kelner and Levin [78] produce a spectral sparsifier with $O(n \log n/\epsilon^2)$ edges in $\tilde{O}(m)$ time. While sparsifiers are a great way of reducing the size of the data, this reduction would then require an additional pass over the data to compute a partitioning which is out of the scope of the problem at hand. Finally, lower bounds are known with regards to the space complexity of both the problem of finding a minimum and maximum cut. Zelke [154] has shown that this cannot be computed in one pass with $o(n^2)$ space.

Finally, analyzing algorithms on random graph models has a long history. In particular, it is quite common to analyze graph partitionings on random graphs [103, 35, 98]. This is done because it is easier to analyze the performance of a partitioning algorithm when we have a clear idea of the 'right' answer.

2.5 The Challenges of One-Pass Streaming Partitioning

The fundamental challenge faced by one-pass streaming algorithms (which are effectively online algorithms) is that they have no idea of what the input behind them looks like. Thus, an early mistake can drastically affect the entire algorithmic input. A clear and understandable example of this is auctioning hotel rooms - if the first person who shows up will pay $100 for either room A or B and you allocate A, then the next person may be willing to pay $500, but only for A. This means, generally, the strategy that these kinds of algorithms employ balances maximizing the pay off today with making sure that each of the possible options is kept available for as long as possible.

Concretely, for streaming partitioning, the problem is even more difficult than the hotel allocation problem. There, when a person arrives, you get to see their entire preference profile. For the graph problem, only part of the preferences are revealed when a vertex arrives, namely we don't see any edges to vertices we haven't already seen in the stream. It is exactly this hiding of information that leads to the lower bounds discussed earlier in the chapter.

The fact that information is hidden by the ordering of the vertices leads to load balancing

problems when considering the balanced partitioning problem. First, if there are strict capacity constraints, i.e. the partitions hold exactly the graph with no slack, then an early mistake can become a big problem later in the process. We are guaranteed to make these early mistakes because of the lack of information, so this justifies the requirement that each partition hold $(1+\epsilon)n/k$ of the graph, even if a perfectly balanced partitioning exists. Second, again, because of the lack of coordination in these types of algorithms, we are guaranteed to make mistakes without enough slack. More precisely, consider a graph that consists of k disconnected equally sized cliques. If this is presented in a random ordering and we try to obtain a k partitioning, we will inevitably try to put two (or more!) of the cliques into the same partitioning initially. As we observe more of the graph, we are able to identify that these are cliques, but since we are trying to grow two or more in the same partition, we will quickly hit the capacity constraint and be forced to partition these cliques.

With strict capacity constraints, 1 mistake early can be a big problem.

Load balancing is hard since it isn't clear what the components are. With k partitions and k components, you run into problems of it tries to concentrate 2 components into 1 partition.

Chapter 3

Experiments on Streaming Graph Partitioning

3.1 Introduction

Given all of the challenges discussed in Chapter 2, the first reasonable approach to the problem of streaming balanced graph partitioning is an experimental one. Does there exist any algorithm that can do well on this problem? A priori, the bounds we can prove on a greedy algorithm only show that it will do no worse than the simple random cut approach. If it is the case that it never does any better, then the correct approach to this problem really is to use the random cut/hashing approach.

In this chapter, we provide a rigorous, empirical study of a set of natural heuristics for streaming balanced graph partitioning. We evaluate these heuristics on a large collection of graph datasets, from various domains: the World Wide Web, social networks, finite-element meshes and synthetic datasets from some popular generative models - preferential-attachment [22], RMAT [90] and Watts-Strogatz [148]. We compare the results of our streaming heuristics to both the hash based partitioning, and METIS [77], a well-regarded, fast, offline partitioning heuristic.

Our results show that some of the heuristics are good and some are surprisingly bad. Our best performing heuristic is a weighted variant of the greedy algorithm. It has a significant improvement over the hashing approach without significantly increasing the computational overhead and obtains an average gain of 76% of the possible improvement in the number of edges cut. On some graphs, with some orderings, a variety of heuristics obtain results which are very close to the offline METIS result. By using the synthetic datasets, we are also able to show that our heuristics scale with the size of the graph and the number of partitions. We demonstrate the value of the best heuristic by using it to partition both the LiveJournal and the Twitter graph for PageRank computation using the Spark cluster system [153]. These are large crawls of real social networks, and we are able to improve the running time of the PageRank algorithm by 18% to 39% by changing the data layout

alone. Our experimental results motivate us to recommend that this is an interesting problem worthy of future research and is a viable preprocessing step for graph computation systems.

Our streaming partitioning is not intended to substitute for a full information graph partitioning. Certain systems or applications that need as good a partitioning as possible will still want to repartition the graph after it has been fully loaded onto the cluster. These systems can still greatly benefit from our optimization as a distributed offline partitioning algorithm started from an already reasonably partitioned graph will require less communication and may need to move fewer vertices, causing it to run faster. Our streaming partitioning algorithms can be viewed as a preprocessing optimization step that cannot hurt in exchange for a very small additional computation cost for every loaded vertex.

3.2 Heuristics and Stream Orders

In this chapter, we examine multiple heuristics and stream orders. We now formally define each one.

3.2.1 Heuristics

The notation P^t refers to the set of partitions at time t. Each individual partition is referred to by its index $P^t(i)$ so $\cup_{i=1}^{k} P^t(i)$ is equal to all of the vertices placed so far. Let v denote the vertex that arrives at time t in the stream, $\Gamma(v)$ refers to the set of vertices that v neighbors and $|S|$ refers to the number of elements in a set S. C is the capacity constraint on each partition. Each of the heuristics gives an algorithm for selecting the index ind of the partition where v is assigned. The first seven heuristics do not use a buffer, while the last three do.

1. Balanced - Assign v to a partition of minimal size, breaking ties randomly:

$$ind = \arg\min_{i \in [k]} \{|P^t(i)|\}$$

2. Chunking - Divide the stream into chunks of size C and fill the partitions completely in order:

$$ind = \lceil t/C \rceil$$

3. Hashing - Given a hash function $H : V \to \{1 \cdots k\}$, assign v to $ind = H(v)$. We use:

$$H(v) = (v \mod k) + 1$$

4. (Weighted) Deterministic Greedy - Assign v to the partition where it has the most edges. Weight this by a penalty function based on the capacity of the partition, penalizing larger partitions. Break ties using Balanced.

$$ind = \arg\max_{i \in [k]} \{|P^t(i) \cap \Gamma(v)| w(t,i)\}$$

where $w(t,i)$ is a weighted penalty function:

$w(t,i) = 1$ for unweighted greedy
$w(t,i) = 1 - \frac{|P^t(i)|}{C}$ for linear weighted
$w(t,i) = 1 - \exp\{|P^t(i)| - C\}$ for exponentially weighted

5. (Weighted) Randomized Greedy - Assign v according to the distribution defined by

$$Pr(i) = |P^t(i) \cap \Gamma(v)| w(t,i)/Z$$

where Z is the normalizing constant and $w(t,i)$ is the above 3 penalty functions.

6. (Weighted) Triangles - Assign v according to

$$\arg\max_{i \in [k]} \{ \frac{|E(P^t(i) \cap \Gamma(v), P^t(i) \cap \Gamma(v))|}{\binom{|P^t(i) \cap \Gamma(v)|}{2}} w(t,i) \}$$

where $w(t,i)$ is the above 3 penalty functions and $E(S,T)$ is the set of edges between the nodes in S and T.

7. Balance Big - Given a way of differentiating high and low degree nodes, if v is high-degree, use Balanced. If it is low-degree, use Deterministic Greedy.

The following heuristics all use a buffer.

8. Prefer Big - Maintain a buffer of size C. Assign all high degree nodes with Balanced, and then stream in more nodes. If the buffer is entirely low degree nodes, then use Deterministic Greedy to clear the buffer.

9. Avoid Big - Maintain a buffer of size C and a threshold on large nodes. Greedily assign all small nodes in the buffer. When the buffer is entirely large nodes, use Deterministic Greedy to clear the buffer.

10. Greedy EvoCut - Use EvoCut [15] on the buffer to find small Nibbles with good conductance. Select a partition for each Nibble using Deterministic Greedy.

Each of these heuristics has a different motivation with some arguably more natural than others. Balanced and Chunking are simple ways of load balancing while ignoring the graph structure.

Hashing is currently used by many real systems [99]. The benefit of Hashing is that every vertex can be quickly found, from any machine in the cluster, without the need to maintain a distributed mapping table. If the IDs of the nodes are consecutive, the hash function $H(v) = (v \mod k) + 1$ makes Balanced and Hashing equivalent. More generally, a pseudorandom hash function should be used, making Hashing equivalent to a random cut.

The greedy approach is standard, although the weighted penalty is inspired by analysis of other online algorithms. The randomized versions of these algorithms were explored because adding randomness can often be shown to theoretically improve the worst-case performance.

The (Weighted) Triangles heuristic exploits work showing that social networks have high clustering coefficients by finding completed triangles among the vertices neighbors in a partition and overweighting their importance.

Heuristics Balance Big, Prefer Big, and Avoid Big assume we have a way to differentiate high and low degree nodes. This assumption is based on the fact that many graphs have power law degree distributions. These three heuristics propose different treatments for the small number of high degree nodes and the large number of low degree nodes.

Balance Big uses the high degree nodes as seeds for the partitions to 'attract' the low degree nodes. The buffered version, Prefer Big, allows the algorithm more choice in finding these seeds. Avoid Big explores the idea that the high degree nodes form the expander portion of the graph, so perhaps the low degrees nodes can be partitioned after the high degree nodes have been removed.

The final heuristic, Greedy EvoCut, uses EvoCut [15], a local partitioning algorithm, on the buffer. This algorithm has very good theoretical guarantees with regards to the found cuts, and the amount of work spent to find them, but the guarantees do not apply to the way we use it.

Edge Balancing While our experiments focus on partitions that are node balanced, in part because this is what our comparator algorithm METIS produces, there is nothing that prevents these heuristics from being used to produce edge-balanced partitions instead, i.e. each partition holds at most $C = (1 + \epsilon)|E|/k$ edges. An edge-balanced partition may be preferable for power-law distributed graphs when the computation to be performed has complexity in terms of the number of edges and not the number of vertices. In fact, in our second set of experiments with the PageRank algorithm in Section 3.5 we used the edge-balanced versions of the algorithms instead, for the above reason.

3.2.2 Stream Orders

In a sense, the stream ordering is the key to having a heuristic perform well. A simple example is Chunking, where, if we had an optimal partitioning, and then created an ordering consisting of all nodes in partition 1, then all nodes in partition 2 and so on, Chunking would also return an optimal partition. For each heuristic, we can define optimal orderings, but, unfortunately, actually generating them reduces to solving balanced graph partitioning so we must settle for orderings that are easy to compute.

We consider the following three stream orderings:

- *Random* - This is a standard ordering in streaming literature and assumes that the vertices arrive in an order given by a random permutation of the vertices.

- *BFS* - This ordering is generated by selecting a starting node from each connected component of the graph uniformly at random and is the result of a breadth-first search that starts at the given node. If there are multiple connected components, the component ordering is done at random.

- *DFS* - This ordering is identical to the *BFS* ordering except that depth-first search is used.

Each of these stream orderings has a different justification. The random ordering is a standard assumption when theoretically analyzing streaming algorithms. While we generate these orderings by selecting a random permutation of the vertices, one could view this as a special case of a generic ordering that does not respect connectivity of the graph. The benefit of a random ordering is that it avoids adversarially bad orderings. The downside is that it does not preserve any locality in the edges so we expect it to do poorly for statistical reasons like the Birthday paradox. Via the Birthday paradox, we can argue that for sparse graphs, we expect to go through $O(\sqrt{n})$ of the vertices before we find a first edge.

Both BFS and DFS are natural ways of linearizing graphs and are highly simplified models of a web crawler. In practice, web crawlers are a combination of local search approaches - they follow links, but fully explore domains and sub-domains before moving on. This is breadth-first search between domains, and depth-first search within. The main benefit of both orderings is that they guarantee that the partitioner sees edges in the stream immediately. Additionally, they maintain some locality. Each has their drawbacks, but it should be noted that BFS is a subroutine that is often used in partitioning algorithms to find a good cut, particularly for rounding fractional solutions to LPs [54].

3.3 Evaluation Setup

We conducted extensive experimental evaluation to discover the performance and trends of stream partitioning heuristics on a variety of graphs. The questions we ask are: Which of these heuristics are reasonable? Can we recommend a best heuristic, restricted to graph type? Do these heuristics scale to larger graphs? Our intent is to use this style of solution for graphs that include trillions of edges, yet in our initial experiments our largest graph has 1.4 million edges. We address this last question by using synthetic datasets to show that the heuristics scale and in Section 3.5 use our heuristics on two larger social networks successfully.

3.3.1 Datasets

We used several sources to collect multiple datasets for our experiments. From the SNAP [89] archive, we used soc-Slashdot0811, wiki-Vote and web-NotreDame. From the Graph Partitioning Archive [147] we used : 3elt, 4elt, and vibrobox. We also used: Astrophysics

collaborations (astro-ph) [110], C. Elegans Neural Network (celegans) [148], and the Marvel Comics social network [9]. We used two large social networks (LiveJournal [105] and Twitter [83]) to evaluate our heuristics in a real system in Section 3.5.

We created synthetic datasets using popular generative models, preferential attachment (BA) [22], Watts-Strogatz (WS) [148], the RMAT generator [90], and a power-law graph generator with clustering (PL) [71]. Three of the synthetic datasets, BA, WS, and PL were created with the NetworkX python package. For each model, we created a degree distribution with average degree $O(\log n)$ (average degree of 10 edges for 1,000 nodes, 13 for 10,000, and 25 for 50,000). This fully specifies the BA model. For WS and PL we used .1 as the rewiring probability. The RMAT datasets were created with the Python Web Graph Generator, a variant of the RMAT generator [1]. The RMAT or Kronecker parameters used by this implementation are [0.45,0.15;0.15,0.25].

The datasets were chosen to balance both size and variety. All are small enough so that we can find offline solutions with METIS so that our results are reproducable, while still big enough to capture the asymptotic behavior of these graph types. The collection captures a variety of real graphs, focusing on finite-element meshes (FEM) and power-law graphs. FEMs are used for scientific computing purposes to model simulations like the flow over a wing, while power-law (and other heavy-tailed) distributions capture nearly all 'natural' graphs, like the World Wide Web, social networks, and protein networks. In general, it is known that FEMs have good partitions because their edges are highly local, while natural graphs are more difficult to partition because they have high expansion and low diameter. The basic statistics about each graph, as well as its type and source are in Table 3.1.

3.3.2 Methodology

We examined all the combinations of datasets, heuristics and stream orders and ran each experiment 5 times on each combination. The *Random* ordering is a random permutation of the vertices, while *BFS* and *DFS* were created by sampling a random vertex to be the root of the BFS or DFS algorithm. Each of the heuristics was run on the same ordering. We ran each experiment on 2, 4, 8, and 16 partitions and fixed the imbalance such that no partition held more than 5% more vertices than its share. The imbalance was chosen as a reasonable setting of this parameter in practice.

3.4 Evaluation Results

In all of the following figures, the y-axis has been scaled to zoom in on the data. The ordering of the heuristics in the figures is the one given in Table 3.2.

Name	$\|V\|$	$\|E\|$	Type	Source
3elt	4720	13,722	FEM	[147]
4elt	15606	45,878	FEM	[147]
vibrobox	12,328	165,250	FEM	[147]
celegans	297	2,148	Protein	[148]
astro-ph	18,772	396,160	Citation	[110]
Slashdot0811	77,360	504,230	Social	[89]
wiki-Vote	7,115	99,291	Social	[89]
Marvel	6,486	427,018	Social	[9]
web-ND	325,729	1,497,134	Web	[89]
BA	1,000	9,900	Synthetic	[22]
BA	10,000	129,831	Synthetic	[22]
BA	50,000	1,249,375	Synthetic	[22]
RMAT	1,000	9,175	Synthetic	[90]
RMAT	10,000	129,015	Synthetic	[90]
RMAT	50,000	1,231,907	Synthetic	[90]
WS	1,000	5,000	Synthetic	[148]
WS	10,000	120,000	Synthetic	[148]
WS	50,000	3,400,000	Synthetic	[148]
PL	1,000	9,878	Synthetic	[71]
PL	10,000	129,763	Synthetic	[71]
PL	50,000	1,249,044	Synthetic	[71]
LiveJournal	$4.6*10^6$	$77.4*10^6$	Social	[105]
Twitter	$41.7*10^6$	$1.468*10^9$	Social	[83]

Table 3.1: Graph datasets summary

3.4.1 Upper and lower bounds

In order to evaluate the quality of our heuristics, we must establish good upper and lower bounds for the performance. A natural upper bound is the approach currently used in practice - hashing the node ID and mapping it to a partition. This approach completely ignores the edges so its expected performance is cutting a $\frac{k-1}{k}$ fraction of edges for k partitions. This bound is marked by the upper black line in our figures. We expect Balanced and Hashing to always perform at this level, as well as Chunking on a random order.

The lower bound can be picked in many more ways. Finding an optimal lower bound is NP-hard, so we focus on more realistic approaches. We compare against a practical and fast approach, the partition produced by METIS v4.0.3. While METIS has no theoretical guarantees, it is widely respected and produces good cuts in practice, and is thus a good offline comparison for our empirical work. This METIS value is marked as the lower black line in our figures. Note that METIS is given significantly more information than the streaming heuristics, so we would not expect them to produce partitioning of the same quality. Any

heuristic between these two lines is an improvement.

3.4.2 Performance on three graph types

We have included figures of the results for three of the graphs, a synthetic graph, a social network graph and a FEM with the goal of covering all three major types of graphs.

Figure 3.1 depicts the performance on the PowerLaw Clustered graph [71] of size 1,000 with 4 partitions. This is one of our synthetic graphs where the model is intended to capture power law graphs observed in nature. The lower bound provided by METIS is 58.9% of the edges cut, while the upper bound for 4 partitions is 75%. The first heuristic, Avoid Big, is worse than a random cut. Linear Deterministic Greedy and Balance Big both perform very well for all 3 stream orderings. These each had a best average performance of 61.7% and 63.2% of the edges cut respectively, corresponding to 82% and 73% of the possible gain in performance. This gain was calculated as the fraction of edges cut by the random minus the fraction cut by the heuristic, divided by the fraction cut by a random cut minus the fraction cut by METIS ($\frac{\text{random}-\text{heuristic}}{\text{random}-\text{METIS}}$).

Figure 3.2 is our results for a social network, the Marvel Comics network [9], with 8 partitions. The Marvel network is synthetic, as it is the result of character interactions in books, but studies have shown it is similar to real social networks. The lower bound from METIS cuts only 32.2% of edges. The upper bound is $7/8 = 87.5\%$ cut. The two heuristics at the upper bound level are Balanced and Hashing, with Chunking on the random order also performing poorly, as expected. Again, the best heuristic is Linear Deterministic Greedy, with 48%, 48.7% and 50.8% edges cut for the *BFS*, *DFS* and *Random* orderings respectively. This constitutes a gain of 71.3%, 70% and 66%.

Figure 3.3 contains the results for a FEM, 4elt [147], with 4 partitions. The change in graph structure gives us quite different results, not the least of which is that METIS now cuts only 0.7% of edges. The upper bound remains at 75%, providing a huge range for improvement. Surprisingly, Chunking performs extremely well here for the *BFS* and *DFS* orders, at 4.7% and 5.7% cut respectively. Translating into gain provides 94.7% and 93.3% of the optimal improvement. Chunking performs poorly on the *Random* order as expected. The other heuristic that performs well is Greedy EvoCut, obtaining 5.1% and 5% cuts for *BFS* and *DFS* respectively. Linear Deterministic Greedy obtains 9.4%, 20.3%, and 30.6% cuts for *BFS*, *DFS* and *Random* respectively. In fact, all of the heuristics beyond Balanced and Hashing are vast improvements. The *BFS* ordering is also a strict improvement for all approaches over the *DFS* and *Random* orderings.

3.4.3 Performance on all graphs: discussion

We present the gain in the performance of each heuristic in Table 3.2, averaged over all datasets from Table 3.1 (except for LiveJournal and Twitter) and all runs, for each ordering. The best heuristic is Linear Deterministic Greedy for all orderings, followed by Balance Big.

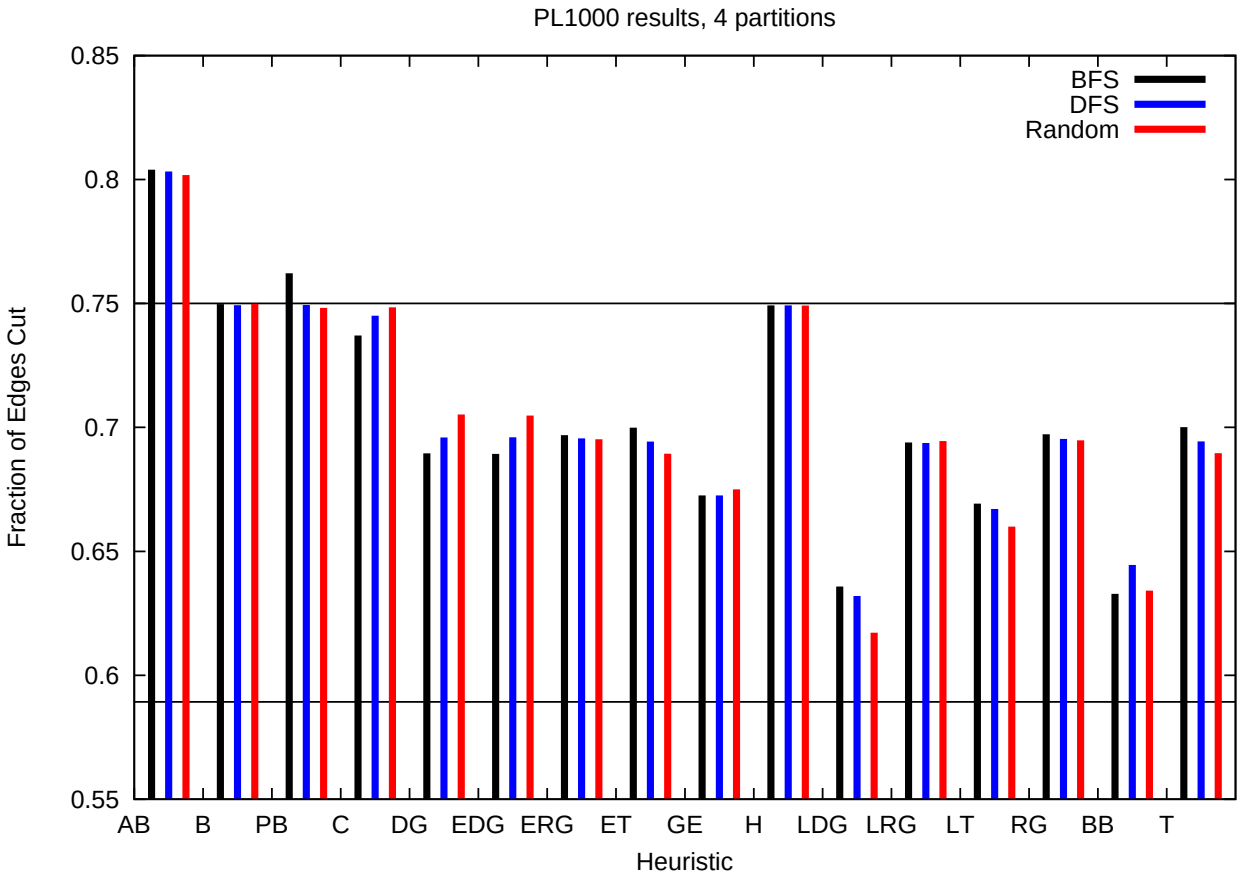

Figure 3.1: PL1000 results. The top line is the cost of a random cut and the bottom line is METIS. The best heuristic is Linear Deterministic Greedy. The figures are best viewed in color.

Greedy EvoCut is also successful on the *BFS* and *DFS* orderings, but is computationally much more expensive than the other two approaches. Note that Balance Big is a combination of the Greedy and Balanced strategies, assigned based on node degree. There are universally bad heuristics, namely Prefer Big and Avoid Big. Both of these are significantly worse than Hashing.

We further restrict the results by type of graphs. As stated earlier, FEMs have good balanced edge cuts. For these types of graphs, no heuristic performed worse than the Hashing approach, and most did significantly better. For the *BFS* ordering, Linear Deterministic Greedy had an average 86.6% gain, with Deterministic Greedy closely behind at 84.2%. For the *DFS* ordering, the Greedy EvoCut approach performed best at 78.8%, with all 3 deterministic greedy approaches closely behind at 74.9% (exp), 74.8% (unweighted) and 75.8% (linear). Finally, as always, the *Random* ordering was the hardest, but Linear Deterministic Greedy was also the best with 63% improvement. No other method achieved more than 56%. The surprising result for FEMs is how well the Chunking heuristic performed: an

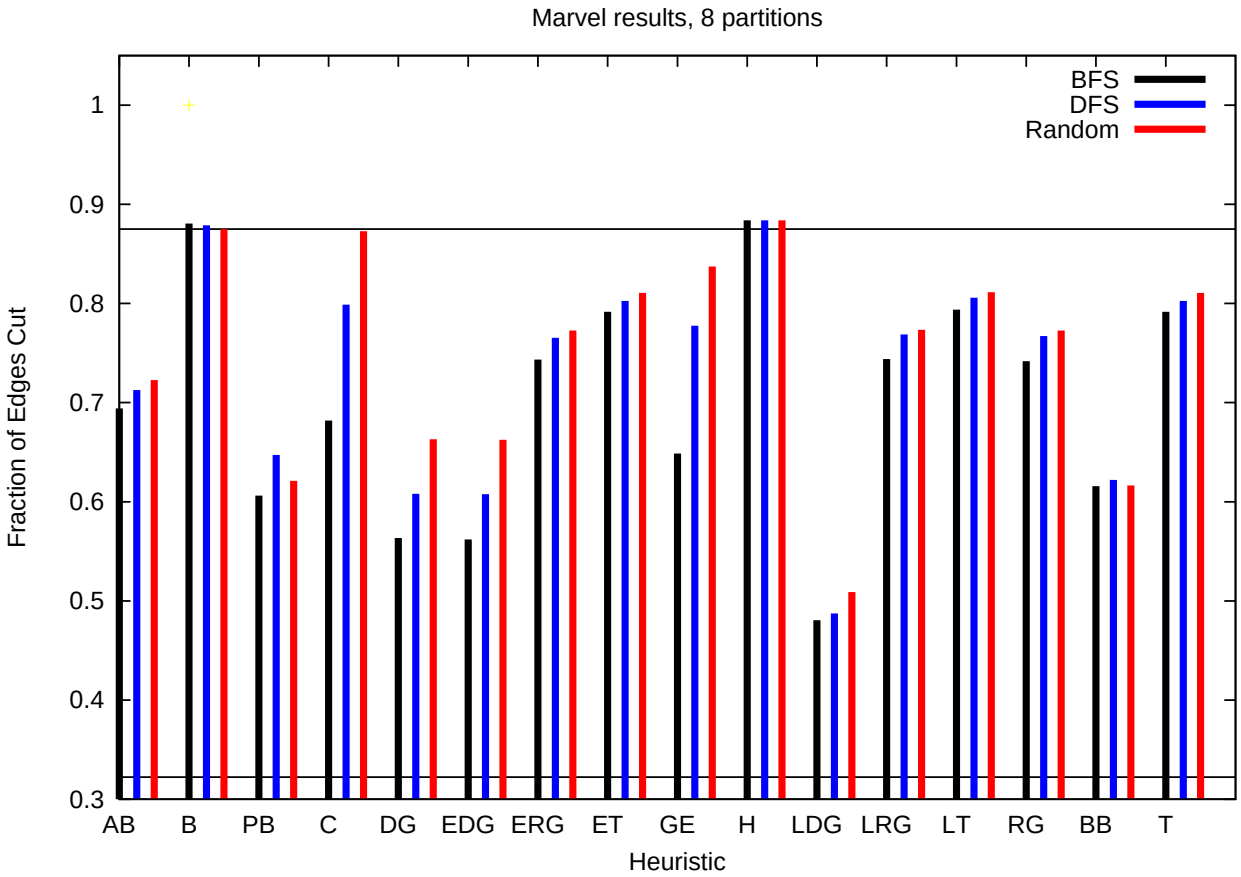

Figure 3.2: Marvel results. The top line is the cost of a random cut and the bottom line is METIS. The best heuristic is Linear Deterministic Greedy.

80% improvement for BFS and 72% for DFS. This is a huge improvement for such a simple heuristic, although it is due to the topology of the networks and the fact that *BFS* is used in partitioning algorithms to find good cuts. When given a *Random* ordering, Chunking had only a 0.2% average improvement.

The social networks results were more varied. Here, the Prefer Big and Avoid Big both have large negative improvements, meaning both should never be used for power law degree networks with high expansion. For all three orderings, Linear Deterministic Greedy was clearly the superior approach with 71% improvement for *BFS*nd 70% for *DFS*. The second best performance was from both Exponential Deterministic Greedy and Deterministic Greedy at 60.5% for *BFS* and 52.9% for *DFS*. Finally, for a *Random* ordering, Linear Deterministic Greedy achieved a 64% improvement, with the other greedy approaches at only 42%.

Given that the Linear Deterministic Greedy algorithm performed so well, even compared with the other variants, one may ask why. At a high level, the penalty functions form a continuum. The unweighted version has a very strict cutoff - the penalty only applies when the partition is full and gives no indication that this restriction is approaching. The

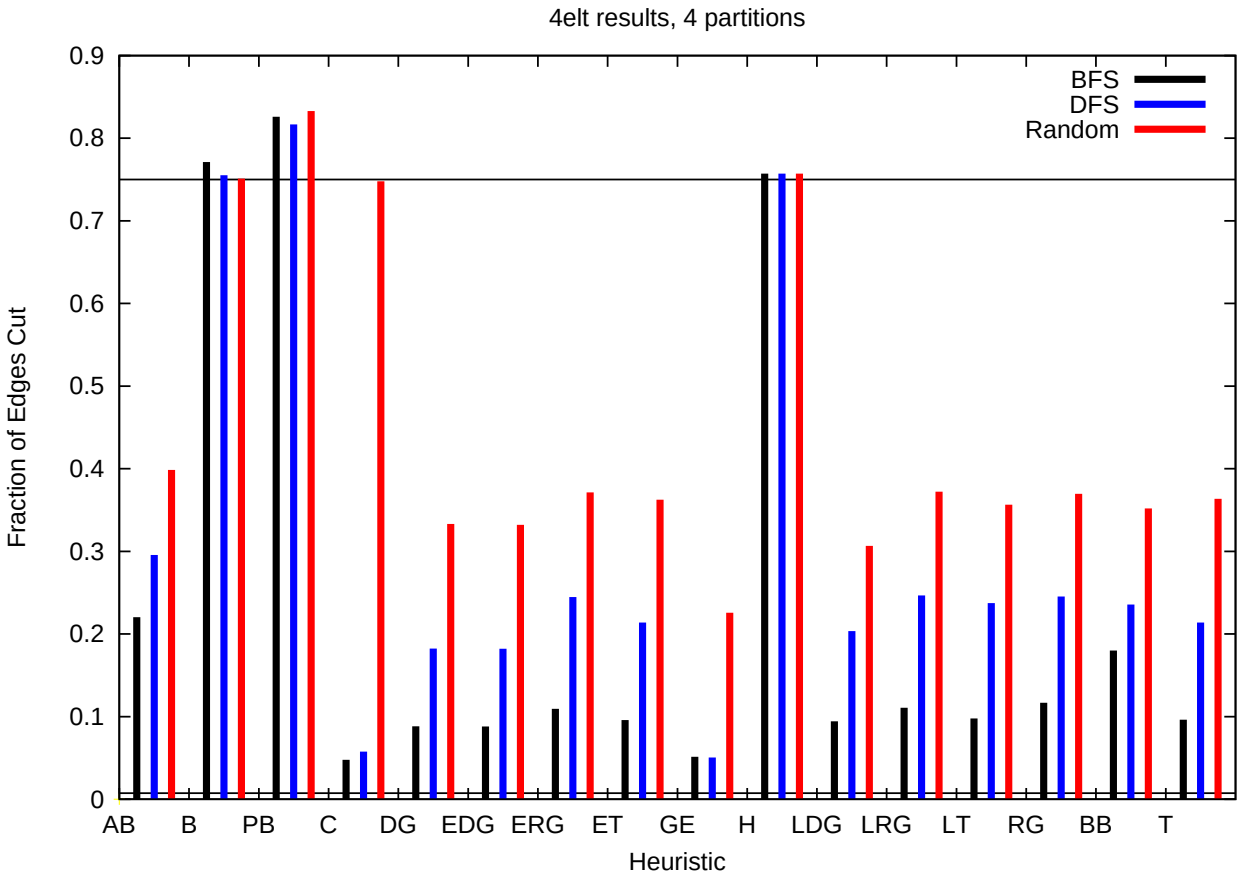

Figure 3.3: 4elt results. The top line is a random cut and the bottom line is METIS (0.7% edges cut).

exponential penalty function has similar performance to the unweighted version because while the exponential function does not indicate that the partition is nearly full until it is very close to the limit. The linear weighting optimally balances the greedy choices with preferring less loaded partitions. Since $1 - x \approx e^{-x}$ when $0 < x < 1$, the linear weighting can be seen as a normalized exponential weighting. This normalization term allows the penalty to take effect much earlier in the process and smooths the information by preventing the size of the partition from affecting the prediction. As this is a continuum, this parameter could be further fine-tuned for different types of graphs. Additionally, the implementation of the unweighted greedy algorithm in this chapter breaks ties lexicographically. Breaking ties by load is equivalent to an indicator penalty function and its performance is very close to the linear penalty function.

Heuristic		*BFS*	*DFS*	*Random*
Avoid Big	AB	-27.3	-38.6	-46.4
Balanced	B	-1.5	-1.3	-0.2
Prefer Big	PB	-9.5	-18.6	-23.1
Chunking	C	37.6	35.7	0.7
Deterministic Greedy	DG	57.7	54.7	45.4
Exp. Det. Greedy	EDG	59.4	56.2	47.5
Exp. Rand. Greedy	ERG	45.6	45.6	38.8
Exp. Triangles	ET	50.7	49.3	41.6
Greedy EvoCut	GE	60.3	58.6	43.1
Hashing	H	-1.9	-2.1	-1.7
Linear Det. Greedy	LDG	76	73	75.3
Linear Rand. Greedy	LRG	46.4	44.9	39.1
Linear Triangles	LT	55.4	54.6	49.3
Randomized Greedy	RG	45.5	44.9	38.7
Balance Big	BB	67.8	68.5	63.3
Triangles	T	49.7	48.4	40.2

Table 3.2: The average gain of each heuristic over all of our datasets and partitions sizes.

3.4.4 Scalability in the graph size

All of our datasets discussed so far are tiny when compared with graphs used in practice. While the above results are promising, it is important to understand whether the heuristics scale with the size of the graph. We used the synthetic datasets in order to control for the variance in different graphs. The key assumption is that using the same generative model with similar parameter settings will guarantee similar graph statistics while allowing the number of edges and nodes to vary. We began by looking at the results for the four generative models, BA, RMAT, WS, and PL. For each of these we had 3 data points: 1,000 vertices, 10,000 vertices, and 50,000 vertices. In order to get a better picture, we created additional graphs with 20,000, 30,000, 100,000 and 200,000 vertices. We will present only the results for the Watts-Strogatz graphs, but all other graphs exhibit quite similar results. Note that these results will scale to any size graph created by the same generative model because of the statistical properties of the generated graphs.

The labels in Figure 3.4 have been elided for clarity of the image. The bottom black line is METIS. This shows that our idea that the fraction of edges cut should scale with the size of the graph holds - it is approximately 12% for each graph. Next, there is clearly a best heuristic for this type of graph, the purple line. It corresponds to the Linear Deterministic Greedy heuristic. It has an average edge cut of 21% over all sizes of the graphs. Finally, all of the lines are approximately constant. The noise in the performance of each algorithm is due to the random nature of the orderings, and would decrease with further trials.

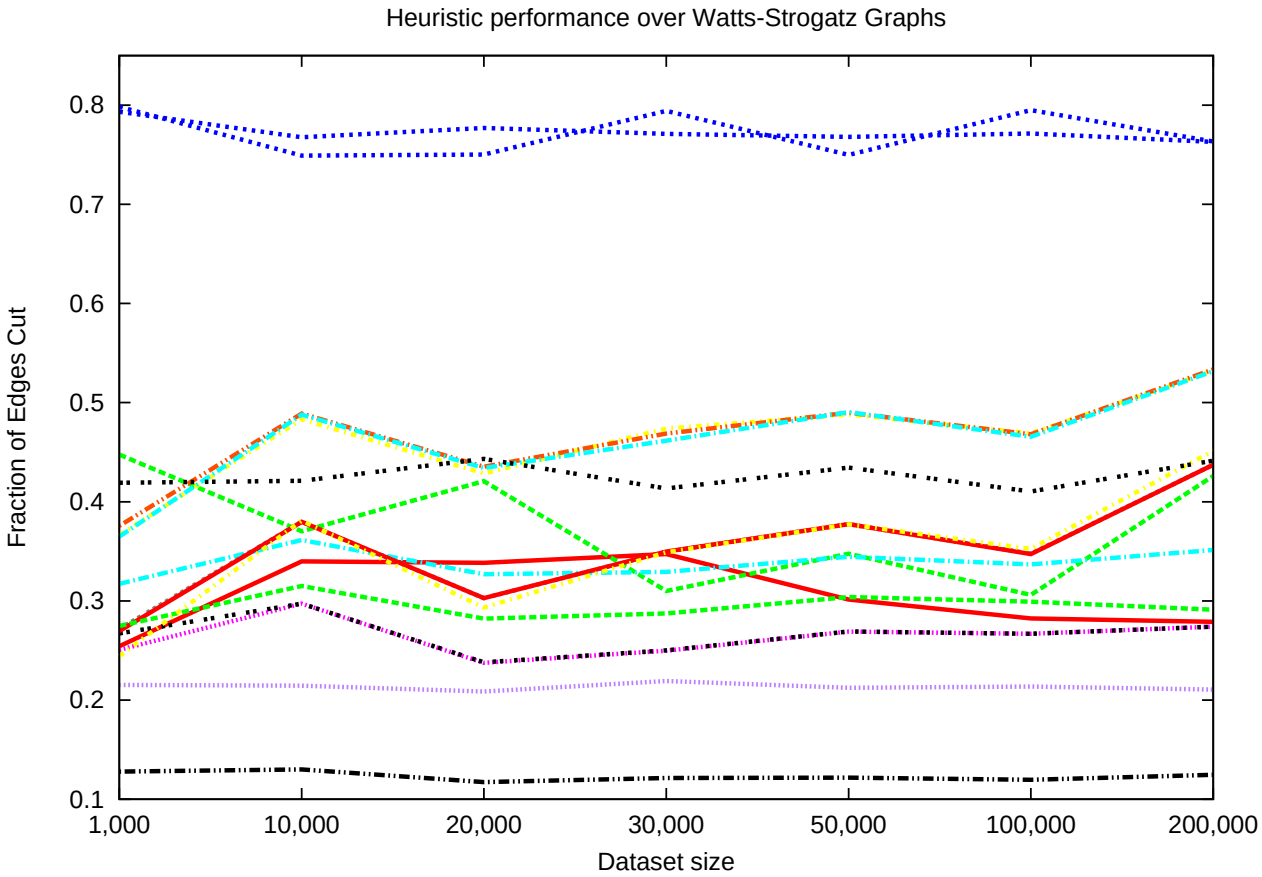

Figure 3.4: *BFS* with 4 partitions. Each line is a heuristics performance over 7 sizes of WS graph. The bottom line is METIS. The bottom purple line is Linear Deterministic Greedy. Best viewed in color.

3.4.5 Scalability in the number of partitions

The other question is how the partitioning quality scales with the number of partitions. The fraction of edges cut must necessarily increase as we increase the number of partitions. Also, we are not trying to find an optimal number of partitions for the graph. As before, we only present data on one graph in Figure 3.5, the 50,000 node PowerLaw Clustered graph, but all graphs have similar characteristics. The heuristics performance closely tracks that of METIS.

3.5 Results on a Real System

After evaluating the performance of the partitioning algorithms, we naturally ask whether the improvement in the partitioning makes any measurable difference for real computations. To evaluate our partitioning scheme in a cluster application, we used an implementation of

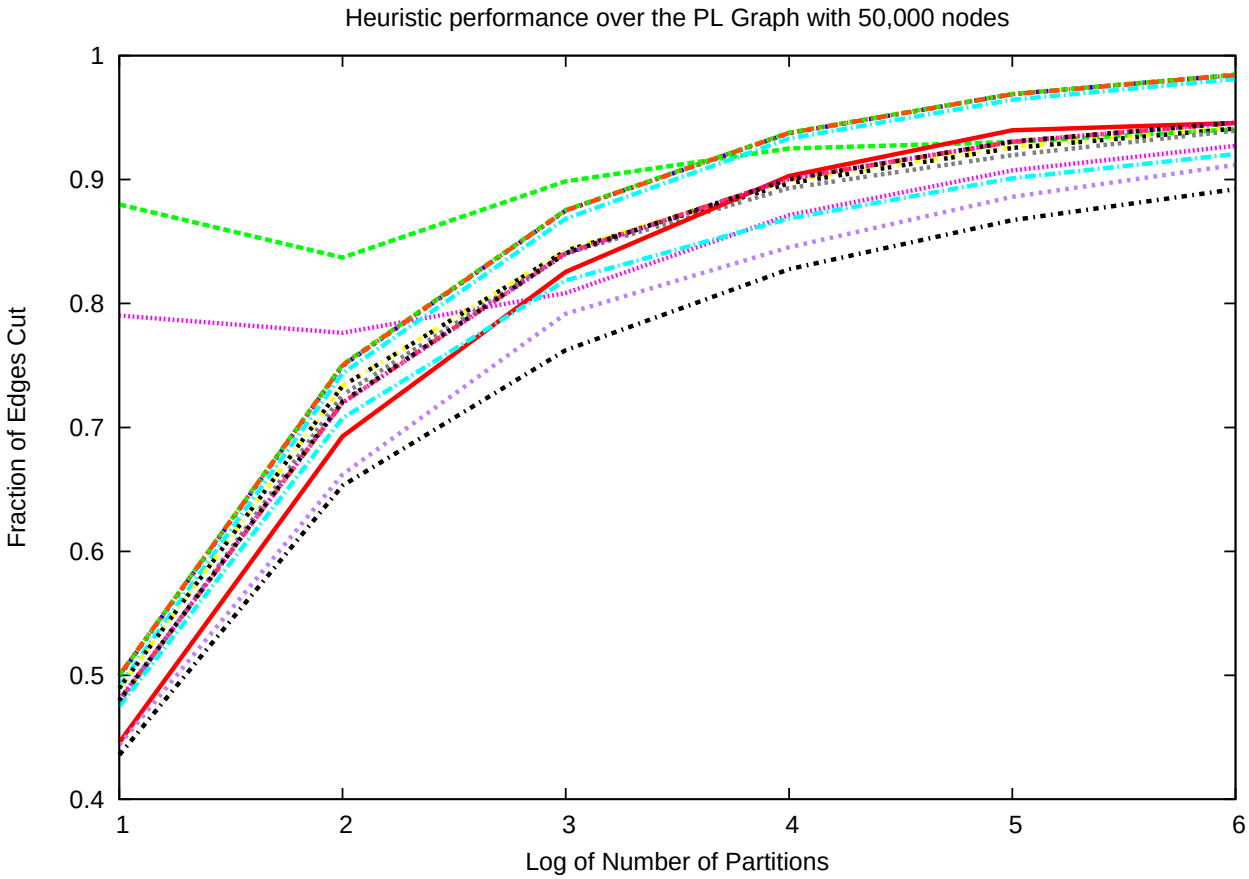

Figure 3.5: *BFS* with 2-64 partitions. Each line connects a heuristics performance over the 6 partition sizes. The bottom line is METIS. The bottom purple line is Linear Deterministic Greedy.

PageRank in Spark [153], a cluster computing framework for iterative applications. Spark provides the ability to keep the working set of the application (the graph topology and PageRank values) in memory across iterations, so the algorithm is primarily limited by communication between nodes. Other recent frameworks for large-scale graph processing, like Pregel [99] and GraphLab [94], also keep data in memory and are expected to exhibit similar performance characteristics.

There are many graph algorithms implemented for the Spark system, but we chose PageRank for two reasons. One is the popularity of this specific algorithm, and the other is its generality. PageRank is a specialized matrix multiplication, and many graph algorithms can be expressed similarly. Additionally, Spark has two implementations of PageRank: a naïve version that sends a message on the network for each edge, and a more sophisticated combiner version that aggregates all messages between each partition [152].

We used Linear Deterministic Greedy, as it performed best in our previous experiments. We tried both a vertex balanced version and an edge-balanced version. However, our datasets

	LJ Hash	LJ Streamed	Twitter Hash	Twitter Streamed
Naïve PR Mean	296.2s	181.5s	1199.4 s	969.3 s
Naïve PR STD	5.5 s	2.2 s	81.2 s	16.9 s
Combiner PR Mean	155.1 s	110.4 s	599.4 s	486.8 s
Combiner PR STD	1.5 s	0.8 s	14.4 s	5.9 s

Table 3.3: Timing data (mean and standard deviation) for 5 iterations of PageRank computation on Spark for LiveJournal and Twitter graphs, Hashing vs. Linear Deterministic Greedy.

are social networks and follow a power-law degree distribution. For PageRank, the quantity that should be balanced is the number of edges in each partition as this controls the amount of computation performed in sparse matrix multiplication and we want this to be equal for all partitions. The existence of very high degree nodes means that some partitions contain many more edges than others, resulting in unbalanced computation times between the different cluster machines. We therefore modified Linear Deterministic Greedy to use the number of edges in a partition for the weight penalty. We used two datasets, LiveJournal [105] with 4.6 million nodes and 77.4 million edges, and Twitter [83] with 41.7 million nodes and 1.468 billion edges. While neither are Internet scale, they are both realistic for medium sized web systems and large enough to show the effects of reduced communication on a distributed computation.

LiveJournal We used 100 partitions, with imbalance of 2% and the stream order provided by the authors of the dataset which is an unknown ordering. Linear Deterministic Greedy reduced the number of edges cut to 47,361,254 edges compared with 76,234,872 for Hashing. We ran 5 iterations of both versions of PageRank, and repeated this experiment 5 times. With the improved partitioning, naïve PageRank was 38.7% faster than the hashed partitioning version.The timing information, along with standard deviations, is summarized in Table 3.3. We used 10 "large" machines (7.5GB memory and 2 CPUs) on Amazon's EC2. The combiner version with our partitioning was 28.8% faster than the hashed version. This reduction in computation time is obtained entirely by laying out the data in a slightly smarter way.

Twitter We repeated the experiment for Twitter [83]. This graph is one of the largest publicly available datasets. Twitter was partitioned into 400 pieces with a maximum imbalance of 2%. Linear Deterministic Greedy cut 1.341 billion edges, while Hashing cut 1.464 billion. We used 50 "large" machines with 100 cores. The total computation time is much longer due to the increase in size. The naïve PageRank was 19.1% faster with our partitioning while the combiner version was 18.8% faster. For both graphs, there was additional time associated with loading the graph, about 200 seconds for Twitter and 80 seconds for LiveJournal, but this was not affected by the partitioning method.

These results show that with very little engineering effort, a simple preprocessing step that considers the graph edges can yield a large improvement in the running time. The best heuristic can be computed for each arriving node in time that is linear in the number of edges, given access to the distributed lookup table for the cluster and knowledge of the current loads of the machines. The improvement in running time is entirely due to the reduced network communication.

3.6 Conclusions

This chapter has demonstrated that simple, one-pass streaming graph partitioning heuristics can dramatically improve the edge-cut in distributed graphs. Our best performing heuristic is the linear weighted variant of the greedy algorithm. This is a simple and effective preprocessing step for large graph computation systems, as the data must be loaded onto the cluster any way. One might need to perform a full graph partitioning once the graph has been fully loaded, however, as it will be re-partitioning an already partitioned graph, there will be less communication cost and it potentially may need to move fewer vertices, and will be faster. Using our approach as preprocessing step can only benefit any future computation while incurring only small cost.

In the next chapter, we will provide a theoretical analysis of both Linear Deterministic Greedy and Randomized Deterministic Greedy. This analysis will clearly show why the performance differences happen.

Chapter 4

Theoretical Results on Streaming Graph Partitioning

Chapter 3 addresses the problem of streaming balanced graph partitioning from an experimental perspectiveand evaluates 16 different partitioning heuristics on 21 different graphs to find how well each performs when compared with an offline partitioning heuristic (METIS [77]). The surprising result was that one of the quite simple heuristics, Linear Deterministic Greedy, performed the best, even beating an adaptation of a local partitioning algorithm, EvoCut [15]. Additionally, adding randomization into the algorithm caused it to perform significantly worse. This is surprising because the addition of randomness often allows us to design more efficient algorithms, not less. In this chapter, we explain theoretically why Linear Deterministic Greedy performed so well and what causes the difference between its performance and that of the randomized variant.

This Chapter focuses on much more stylized algorithms and maps them to random processes. As a result, to separate the two chapters, we will call Linear Deterministic Greedy the arg max Greedy algorithm in this chapter, and Linear Randomized Greedy will be the Proportional Greedy algorithm.

Contributions This chapter focuses on developing a rigorous understanding of two greedy streaming balanced graph partitioning algorithms. We first give lower bounds on the approximation ratio that any streaming algorithm for balanced graph partitioning can obtain on both a random and adversarial ordering of the graph. In response to this lower bound, we focus our attention on a class of random graphs with embedded balanced k cuts. We analyze our greedy algorithms by using a novel coupling to finite Polya Urn processes. This is very elucidating connection gives clear intuition as to why one algorithm performs well while the other does not.

4.1 Notation and Definitions

We now introduce the notation and definitions used throughout the rest of the paper. The *balanced graph partitioning problem* takes as input a graph G, an integer k and an allowed imbalance parameter of ϵ. The goal is to partition the vertices of G into k sets, each no larger than $(1+\epsilon)\frac{n}{k}$ vertices, while minimizing the number of edges cut.

Graph Models A graph $G = (V, E)$ consists of $n = |V|$ vertices and $m = |E|$ edges. $\Gamma(v)$ is the set of vertices that a vertex v neighbors. We consider graphs generated by two random models. The first, $G(n, p)$ is the traditional Erdös-Renyi model with n vertices. The traditional definition is that each of the possible $\binom{n}{2}$ edges is included independently with probability p. At certain points in the proofs in Section 5, we modify this definition to make it better match our streaming model. In particular, we allow multiple edges in order to maintain independence in our analysis.

$G(\Psi, P)$ is a generalization of $G(n, p)$, due to McSherry [103], that allows the graph to have l different Erdös-Renyi components, each with different parameters. Again, we have n vertices. $\Psi : \{1, 2, \ldots n\} \to \{1, 2, \ldots l\}$ is a function mapping the vertices into l disjoint clusters. Let C_i refer to the set of vertices mapped to i, i.e. $\Psi^{-1}(i) = C_i$. P is a $l \times l$ matrix where edges between vertices in C_i are included independently with probability $P_{i,i}$ and edges between vertices in C_i and C_j are included with probability $P_{i,j}$. There are many ways for $G(\Psi, P)$ to generate graphs in $G(n, p)$ - Ψ could map all vertices into the same cluster or we could have $P_{i,i} = P_{i,j} = p$ for all i, j. We make the same modification to the generative process as in $G(n, p)$ and allow multiple edges for clarity of the analysis.

Probability Distributions We only use variables drawn from a binomial distribution, where $X \sim B(n, p)$ is a random variable representing n independent trials, each with probability p of success.

4.1.1 Polya Urn Processes

The classical Polya Urn problem is: Given finitely many initial bins, each containing one ball, let additional balls arrive one at a time. For each new ball with probability p create a new bin and put the ball in it. With probability $1 - p$, place the ball in an existing bin with probability proportional to m^γ where m is the number of balls currently in that bin.

Many variants of the above process have been analyzed. In particular, Chung, Handjani, and Jungreis [38] analyze the *finite Polya urn process* where $p = 0$. The exponent γ plays an important role in the behavior of this process. With k bins, when $\gamma < 1$, in the limit, the load of each bin is uniformly distributed and each contains a $\frac{1}{k}$ fraction of the balls. When $\gamma > 1$, in the limit, the fractional load of one bin is 1. When $\gamma = 1$, the limit of the fractional loads exists but is distributed uniformly on the simplex.

Our proof technique will focus on connecting the streaming graph partitioning algorithms with the finite Polya urn process and use many of the results from [38]. We restate the results used here:

Theorem 3 (Theorem 2.1 from [38]). *Consider a finite Polya process with exponent $\gamma = 1$, k bins and let x_i^t denote the fraction of balls in the i^{th} bin at time t. Then almost surely for each i, the limit $X_i = \lim_{t\to\infty} x_i^t$ exists. Furthermore these limits are distributed uniformly on the simplex $\{(X_1, X_2, \ldots X_k) : X_i > 0, X_1 + X_2 + \ldots + X_k = 1\}$.*

Theorem 4 (Theorem 2.2 from [38]). *Consider a finite k-bin Polya process with exponent γ and let x_i^t denote the fraction of the balls in bin i at time t. Then a.s. the limit $X_i = \lim_{t\to\infty} x_i^t$ exists for each i. If $\gamma > 1$ then $X_i = 1$ for one bin, and $X_i = 0$ for all others. If $\gamma < 1$ then $X_i = \frac{1}{k}$ for all bins.*

Lemma 1 (Lemma 2.3 from [38]). *Given a finite or infinite Polya process with exponent γ and an arbitrary initial configuration (i.e. finitely many balls arranged in finitely many bins), suppose we restrict attention to any particular subset of the bins and ignore any balls that are placed in the other bins. Then the process behaves exactly like a finite Polya process with exponent γ on this subset of bins, though the process may terminate after finitely many balls.*

Lemma 1 is particularly important to our analysis as it forms the basis of an inductive argument to extend the analysis in [38] to k bins from 2 bins. We also use the claim that a finite, arbitrary initial configuration does not affect the distribution in the limit.

4.1.2 The Streaming Model

We consider a streaming graph model where the vertices arrive in some order. The two stream orderings we consider are *adversarial* and *random*. For n vertices, the set of permutations S_n defines all possible orderings. For a random ordering, each permutation is picked with equal probability. An adversarial ordering is any probability distribution over the permutations, including one that picks the worst possible ordering for the algorithm.

When a vertex arrives so do all of its incident edges. Our goal is to generate a balanced vertex partitioning of the graph with k partitions. The capacity of each partition, C, is enough to hold all the vertices, i.e. $kC = (1+\epsilon)n$. We assume an undirected graph since our evaluation metric, the number of edges cut, is not affected by the directionality of an edge.

We chose this model because we are concerned with the problem of loading data onto a cluster and partitioning at the same time. We assume that only one pass can be made over the data and the algorithm has access to the current load of each machine on the cluster and the location of each vertex that has been previously seen. A vertex is not moved after it has been placed into some partition.

4.2 Lower Bounds

Given our streaming model, the first important question is whether any algorithm can do well on all graphs. The unfortunate answer is no. Intuitively, with only one pass, important edges may be hidden either intentionally by an adversary or unintentionally by randomness.

Theorem 5. *One-pass streaming balanced graph partitioning with an adversarial stream order can not be approximated within $o(n)$.*

Proof. Without loss of generality, we seek a balanced 2 partitioning. Consider a graph that is a cycle over n vertices with edges such that $(i, i+1) \mod n \in E$ for $1 \leq i \leq n$. Let the ordering be all odd nodes, then all even, i.e. $1, 3, 5 \ldots n-1, 2, 4, 6 \ldots n$. Assume that n is even. The optimal balanced partitioning cuts 2 edges. However, the given ordering reveals no edges until $\frac{n}{2}$ vertices arrive. Until the edges arrive, we have no way of distinguishing which vertices are 'near' each other. In particular, note that this ordering is indistinguishable from one where the odd vertices are given in a random order, or one where the odd nodes are interspersed with unconnected even nodes, i.e. $1, n-2, 3, n-4, 5, n-6 \ldots$. Thus, no algorithm can do better than cutting $\frac{n}{2}$ edges in expectation. This generalizes to k partitions. □

Theorem 6. *One-pass streaming balanced graph partitioning with a random stream order can not be approximated within $o(n)$.*

Proof. Again, we seek a balanced 2 partition for a cycle graph with a random ordering. Consider the t^{th} vertex to arrive in this ordering.

$$\begin{gathered}\Pr t \text{ arrives with no edges } = \\ \Pr \text{ both neighbors arrive after } t = \frac{n-t}{n}\frac{n-t-1}{n-1}\end{gathered}$$

so the number of vertices that we expect to arrive with no edges is

$$\mathbf{E}[\#\text{ with no edge}] = \sum_{t=1}^{n} \frac{t}{n}\frac{t+1}{n-1} \approx \frac{1}{n^2}\sum_{t=1}^{n} t^2 - t = \frac{1}{n^2}\left(\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} + \frac{n(n+1)}{2}\right)$$

Therefore, asymptotically, we expect $\frac{n}{3}$ vertices to arrive with no edges. As before, when a vertex arrives with no edges, we are not able to determine which other vertices it is 'near'. For each of these, we expect to cut 1 edge, providing us with our lower bound. □

In the following sections, we only analyze the algorithms for random orderings. In particular, we will show that for random graphs with higher degree and a planted partition, arg max Greedycan recover the partitioning.

4.3 Analysis of Algorithms on Random Graphs

The experiments in [136] showed that one heuristic studied in the paper, Linear Deterministic Greedy (LDG), was clearly the best tried. However, another heuristic, Linear Randomized Greedy(LRG), differs only in that it selects a partition proportionally to the distribution

Algorithm 1 arg max Greedy

Input: G, k, C, π

```
P_1, ··· , P_k = ∅
for t = 1, 2, ... n do
    for i = 1, 2, ... k do
        S_i = |Γ(π(t)) ∩ P_i|
        if |P_i| = C then
            S_i = 0
    if all S_i = 0 then
        Pick i from arg min_{j∈[k]}{|P_j|} u.a.r.
    else
        Pick i from arg max_{j∈[k]}{S_j} u.a.r.
    P_i = P_i ∪ π(t)
```

Algorithm 2 Proportional Greedy

Input: G, k, C, π

```
P_1, ··· , P_k = ∅
for t = 1, 2, ... n do
    for i = 1, 2, ... k do
        S_i = |Γ(π(t)) ∩ P_i|
        if |P_i| = C then
            S_i = 0
    if all S_i = 0 then
        Pick i from arg min_{j∈[k]}{|P_j|} u.a.r.
    else
        Pick i proportional to S_i
    P_i = P_i ∪ π(t)
```

of edges instead of from the maxima. In the experiments, LDG performed significantly better than LRG. This raises the question - can we theoretically explain the difference in performance? In this section, we will introduce slightly simpler variants, arg max Greedy (corresponding to LDG) and Proportional Greedy(corresponding to LRG), and analyze their performance on McSherry's random graph model. Our analysis will clearly demonstrate the difference observed in the experiments.

4.3.1 Algorithms

The two algorithms studied in this paper are very similar: when a vertex v arrives, a score for each partition P_i of the number of edges from v to P_i, $S_i = |\Gamma(v) \cap P_i|$, is calculated. If the partition is full, its score is set to 0. If all scores are 0, then the vertex is assigned to some

Algorithm 3 arg max Greedy Process on $G(n, p)$

Input: p

Set $P_1, P_2, \ldots P_k = \emptyset$
for $t = 1, 2, \ldots n$ **do**
  For $1 \leq i \leq k$, draw $E_i^{(t)} \sim B(|P_i|, p)$
  if $\sum_{i=1}^{k} E_i^{(t)} = 0$ **then**
    Assign t to $\arg\min_{j \in [k]}\{|P_j|\}$
  else
    Assign t to $\arg\max_{j \in [k]}\{E_j^{(t)}\}$

partition with minimal load. If a score is non-zero, then the arg max Greedy Algorithm assigns the vertex uniformly at random to a partition in $\arg\max S_i$. By contrast, the Proportional Greedy Algorithm uses the scores as a distribution, assigning the vertex to partition i with probability $S_i / \sum S_j$.

The versions of these algorithms from [136] differ only in that the score for each partition is weighted by the current load of the partition, i.e. $S_i(1 - \frac{|P_i|}{C})$. In practice, the algorithms keep the partitions nearly balanced, meaning this tiebreaker is only used in cases of tied number of edges and when there are no edges where [136] prefers the least-loaded partitions.

One of the key insights of this paper is that when these algorithms are used on random graphs, we can write both down as random processes. In particular, we can let the random process generate the graph while also partitioning it at the same time. This reduction will be discussed in Section 4.3.3. The proof proceeds by analyzing the random process versions of the algorithms, rather than those given in Algorithms 1 and 2.

The random processes generate a multi-edge $G(n, p)$ graph. For the extended $G(\Psi, P)$ analysis, we will only consider Algorithm 1 and the correctly modified version of Algorithm 3. The modification is only with the generation of the $E_i^{(t)}$ and will be discussed later.

4.3.2 Result and Proof Outline

The rest of the paper will focus on proving the following two statements. The first is that the Proportional Greedy Algorithm can not recover an embedded partition in a $G(\Psi, P)$ graph, no matter what the parameters are or how big the graph is. By contrast, the second result is that the arg max Greedy Algorithm can recover the embedded partition, provided the components are dense enough, the cut between them is sparse enough, and there are enough components.

Theorem 7. *Let p be the probability of edges within components and q be the probability of edges between components. Given a $G(\Psi, P)$ graph with $l > k \log k$ equally sized components where $p > \frac{2 \log n}{|C|}$, $p > 3(k + \sqrt{k} + 1)lq$, and $q = O((k^{2.4} \log l)^{-1})$, arg max Greedy Algorithm will recover an embedded partition from a random stream ordering.*

Algorithm 4 Proportional Greedy Process on $G(n,p)$

Input: p

Set $P_1, P_2, \ldots P_k = \emptyset$
for $t = 1, 2, \ldots n$ **do**
For $1 \leq i \leq k$, draw $E_i^{(t)} \sim B(|P_i|, p)$
if $\sum_{i=1}^{k} E_i^{(t)} = 0$ **then**
Assign t to $\arg\min_{j \in [k]}\{|P_j|\}$
else
Assign t to P_i with probability $E_i^{(t)} / \sum_{j=1}^{k} E_j^{(t)}$

The proof proceeds in several stages. First, we ignore the capacity constraint and consider Algorithms 3 and 4 on a single $G(n,p)$ component. Does the algorithm eventually learn it is a component and place it in the same partition? We show that Algorithm 4 is equivalent to a finite Polya urn process with $\gamma = 1$ and distributes the component over all the partitions. By contrast, Algorithm 3 can be coupled to a finite Polya urn process with $\gamma > 1$. It will asymptotically place the entire $G(n,p)$ component in one partition. This argument starts with 2 partitions and is extended to k bins using an induction argument.

That Algorithm 3 will correctly (not) partition a connected component forms the basis of our argument that it can be extended to the $G(\Psi, P)$ model. Intuitively, with the correct parameters, each component of $G(\Psi, P)$ will be placed in a single partition. The primary technical difficulties faced are the inclusion of the capacity constraint, requiring bounds on the component sizes, and the addition of intra-cluster edges, which serve to 'confuse' the algorithm about to which component a vertex belongs. By setting the parameters of the model correctly, we can overcome these challenges.

4.3.3 Analysis on a Single $G(n,p)$ Component

We now analyze Algorithms 3 and 4. These are obtained from Algorithms 1 and 2 by considering the process in terms of Polya urns. As a reminder, the finite Polya urn process has k bins and the t^{th} ball is assigned to bin i with probability proportional to $(m_i^{(t)})^\gamma$ where $m_i^{(t)}$ is the load of the i^{th} bin at time t.

Translating Algorithm 1 and 2 to Polya Urn processes involves identifying each ball with a vertex and each bin with a partition. There are two primary differences from the standard Polya Urn process. First, with probability $(1-p)^t$, the t^{th} vertex (ball) does not have edges to vertices already seen and it is placed in the least loaded partition (urn). The second is that we do not assign the vertex (ball) based on the load of the partition (urn) but instead on a binomial random variable based on the load. Specifically, let $E_1^{(t)}, \ldots E_k^{(t)}$ be the random variables representing the number of edges to each of the k partitions. Each $E_i^{(t)}$ is drawn from $B(m_i^{(t)}, p)$. The following connection is how we created Algorithms 3 and 4.

- Algorithm 1 assigns the vertex to a partition in $\arg\max_{j\in[k]}\{E_j^{(t)}\}$, breaking ties at random.
- Algorithm 2 assigns it to bin i proportional to $E_i^{(t)}$

Algorithm 4 Analysis. Consider the total number of edges from vertex t as a random variable $E^{(t)} \sim B(t,p)$. Each edge is distributed according to $m_i^{(t)}$ i.e. with probability $\frac{m_i^{(t)}}{t}$ it connects to the i^{th} bin. Each of the $E^{(t)}$ edges are distributed i.i.d. and are given equal weight so Algorithm 2 assigns balls proportional to $(m_i^{(t)})^\gamma$ where $\gamma = 1$.

Theorem 8. ***Algorithm 4 on*** $G(n,p)$ *Let* $0 \le p < 1$. *Let* x_i^t *be the fractional load of partition* i *at time* t *of Algorithm 4. Then almost surely* $\lim_{t\to\infty} x_i^t = X_i$ *exists and for all* i, $X_i > 0$.

Proof. We show that when there are edges, this process is exactly a finite Polya urn process with $\gamma = 1$. The result then follows directly from Theorem 3. Let there be k bins. At time t, each has load $m_i^{(t)}$. Let $E^{(t)}$ be the total number of edges drawn by the process. Assume $E^{(t)} > 0$ as $E_t^{()} = 0$ will be dealt with later. Recall that we allow multiple edges in our model, so consider the edges being distributed to the k partitions with replacement, i.e. each of the $E^{(t)}$ edges goes to partition i with probability $\frac{m_i^{(t)}}{t-1}$. Let $E_i^{(t)}$ be the number to partition i. Note that $\sum_{i=1}^k E_i^{(t)} = E^{(t)}$. Now Pr Algorithm 4 picks bin $i = E_i^{(t)}/E^{(t)}$. However, $E_i^{(t)} \sim B(E^{(t)}, \frac{m_i^{(t)}}{t-1})$, showing that this assignment is proportional to $m_i^{(t)}$ as desired. This is exactly a finite Polya urn process with $\gamma = 1$.

The remaining details concern the modification of the process when $E^{(t)} = 0$. In this case, the algorithm will assign the vertex to the least loaded bin. If this situation has a constant probability throughout the process, then it is making the distribution of the balls more uniform, and satisfy the theorem statement that all bins contain a non-zero fraction of the balls. If it is the case that this becomes unlikely as the process progresses, i.e. $p > \frac{\log n}{n}$, then we can apply Theorem 3 and Lemma 1 from [38] to say that after $O(\frac{\log n}{p})$ vertices have arrived, we begin the $\gamma = 1$ Polya Urn process with an arbitrary finite initial configuration. From Theorem 3, we get that $X_i > 0$ for all i. □

We conclude that the randomized algorithm does not have a concentration result. No matter the value of p or the size of the graph, for a $G(n,p)$ component, the Proportional Greedy algorithm will not learn that it is a component and instead distributes it over all partitions.

Corollary 1. *Given a single isolated* $G(n,p)$ *component, for any value* p, *Algorithm 2 will distribute this component over all* k *partitions.*

Algorithm 3 Analysis. The key insight about why Algorithm 3 provides a concentration result is that by preferring the arg max of the distribution of edges, once some partition has

a slightly higher load than the other it is very likely to be assigned the next vertex. As the gap in the loads grow, the larger partition becomes increasingly more likely to receive the next vertex until it is impossible for the smaller partition to compete. However, there are a few challenges.

The first is that with probability $(1-p)^t$ the $t+1^{th}$ vertex will not have any edges to previously seen vertices. In this case, it is automatically placed in the least loaded bin. When this happens, it decreases the gap in the loads. If it happens too often, the gap will not grow. Since $(1-p)^t \approx e^{-pt}$, once $t = O(\frac{\log n}{p})$, this does not happen with high probability, provided $p > \frac{\log n}{n}$. We only expect $\frac{1}{p}$ vertices to arrive with no edges and they are concentrated at the beginning of the process when $t < \frac{1}{p}$.

The second challenge is that when the vertex has 1 edge, the arg max distribution is the same as Algorithm 4. However, this can be dealt with in the same manner as having no edges. Again, we expect $\frac{1}{p}$ vertices to have only 1 edge and primarily when $\frac{1}{p} \leq t \leq \frac{2}{p}$. Therefore, we need $p > \frac{2\log n}{n}$.

The final challenge is that we are not be able to couple Algorithm 3 to a finite Polya urn process with $\gamma > 1$ until $\frac{2}{p}$ vertices have arrived, meaning we do not start with a uniform load distribution. Lemma 1 shows that we can start with an arbitrary finite initial configuration and obtain the same concentration results.

Theorem 9. *Let p be any value between $\frac{2\log n}{n}$ and 1. Let x_i^t be the fractional load of partition i at time t of Algorithm 3. Then almost surely $\lim_{t\to\infty} x_i^t = X_i$ exists and one $X_j = 1$, while all others are 0.*

This statement follows from Theorem 4. Our analysis for Algorithm 3 relies on the probability that bin i will receive a ball at time t or

$$\Pr E_i^{(t)} = \arg\max_{j\in[k]}\{E_j^{(t)}\}$$

for $E_i^{(t)} \sim B(m_i^{(t)}, p)$. It is intuitive that bins with a higher load should have a much higher probability of being the arg max, yet the binomial distribution does not have a nice closed form expression for $\Pr X \geq k$. Even if we condition on $E^{(t)} = \sum_{i=1}^k E_i^{(t)} = x$ so we can express the $E_i^{(t)}$ as a multinomial distribution, a nice closed form solution eludes us.

Therefore, our proof consists of several lemmas.

Lemma 2. *Given a $G(n,p)$ graph with $p > \frac{2\log n}{n}$, after $O(\frac{\log n}{p})$ steps, Algorithm 3 with 2 partitions can be coupled to a finite Polya urn process with $\gamma > 1$.*

Proof. Let $A = E_1^{(t)}$ and $B = E_2^{(t)}$ and A^j, B^j be the loads conditioned on the fact that $E^{(t)} = j$ i.e. $A^j + B^j = j$. Let δ be the comparative advantage of A over B, i.e. $\frac{1}{2} + \delta = \frac{m_1^{(t)}}{t}$ and $\frac{1}{2} - \delta = \frac{m_2^{(t)}}{t}$. We want to analyze $\Pr A^j > B^j$.

$$\Pr A^j > B^j = \sum_{i=\lfloor j/2 \rfloor + 1}^{j} \binom{j}{i} (\frac{1}{2} + \delta)^i (\frac{1}{2} - \delta)^{j-i}$$

$$= (\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor + 1} \sum_{i=\lfloor j/2 \rfloor + 1}^{j} \binom{j}{i} (\frac{1}{2} + \delta)^{i - \lfloor j/2 \rfloor - 1} (\frac{1}{2} - \delta)^{j-i}$$

$$= (\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor + 1} \sum_{i=0}^{\lfloor j/2 \rfloor} \binom{j}{i - \lfloor j/2 \rfloor} (\frac{1}{2} + \delta)^{i} (\frac{1}{2} - \delta)^{j - \lfloor j/2 \rfloor - 1 - i}$$

$$= (\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor + 1} \sum_{i=0}^{\lfloor j/2 \rfloor} \binom{j}{i} (\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor - i} (\frac{1}{2} - \delta)^{i}$$

We similarly express $\Pr B^j > A^j$ as follows.

$$\Pr B^j > A^j = \sum_{i=\lfloor j/2 \rfloor + 1}^{j} \binom{j}{i} (\frac{1}{2} - \delta)^i (\frac{1}{2} + \delta)^{j-i}$$

$$= (\frac{1}{2} - \delta)^{\lfloor j/2 \rfloor + 1} \sum_{i=0}^{\lfloor j/2 \rfloor} \binom{j}{i} (\frac{1}{2} - \delta)^{\lfloor j/2 \rfloor - i} (\frac{1}{2} + \delta)^{i}$$

Because $\frac{1}{2} + \delta > \frac{1}{2} - \delta$, we have that $\sum_{i=0}^{\lfloor j/2 \rfloor} \binom{j}{i} (\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor - i} (\frac{1}{2} - \delta)^i > \sum_{i=0}^{\lfloor j/2 \rfloor} \binom{j}{i} (\frac{1}{2} - \delta)^{\lfloor j/2 \rfloor - i} (\frac{1}{2} + \delta)^i$. Therefore,

$$\Pr A^j > B^j > \frac{(\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor + 1}}{(\frac{1}{2} - \delta)^{\lfloor j/2 \rfloor + 1}} \Pr B^j > A^j.$$

From this, and the fact that these two quantities sum to 1, we conclude that

$$\Pr A^j > B^j > \frac{(\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor + 1}}{(\frac{1}{2} + \delta)^{\lfloor j/2 \rfloor + 1} + (\frac{1}{2} - \delta)^{\lfloor j/2 \rfloor + 1}}$$

This lower bound is the probability that the ball goes in urn 1 in a Polya process with $\gamma = \lfloor j/2 \rfloor + 1$. When $j \geq 2$, we can couple our process to a finite Polya urn process with a desirable concentration result. We remove the conditioning on $E^{(t)} = j$ to get $\Pr A > B$

$$\Pr A > B = \sum_{j=1}^{t} \binom{t}{j} p^j (1-p)^{t-j} \Pr A^j > B^j \tag{4.1}$$

The only case where we are mixing in a process that has an undesirable exponent ($\gamma = 1$) is when $j = 0$ or 1. The probability of this case is less than $\frac{1}{n}$ when $t > \frac{2 \log n}{p}$. According to Lemma 1, this constitutes a finite arbitrary configuration and the concentration results hold after $t > \frac{2 \log n}{p}$. □

The above proof shows that, at some point, the algorithm can be coupled with a finite Polya urn process with $\gamma > 1$. However, we need Lemma 1 from [38] to show that the initial configuration when the process takes off does not affect the concentration results. Moreover, we bound the total expected number of vertices to arrive with $j = 0$ or 1 by

$$\frac{1-(1-p)^n+1-p}{p} \approx \frac{2-e^{-pn}-p}{p} \leq \frac{2}{p}.$$

Combining Lemma 1 and 2 shows that for 2 partitions Algorithm 3 will concentrate the process into 1 bin. In order to extend the process to k partitions, we present the following Lemma. It follows the proof technique of Theorem 4 in [38] and utilizes Lemma 1

Lemma 3. *Consider Algorithm 3 with k partitions on a $G(n,p)$ graph with $p > \frac{2\log n}{n}$. Let x_i^t be the fractional load of the i^{th} partition at time t. Then a.s. the limit $X_i = \lim_{n,t\to\infty} x_i^t$ exists for each i. For exactly one i, $X_i = 1$.*

Proof. To extend the analysis of Lemma 2 from 2 partitions to k, we use induction and condition on each pair of bins. Of the k bins, select 2 and call them A and B. We modify Lemma 2's Equation 4.1 by substituting

$$\Pr E^{(t)} = j = \binom{t}{j} p^j (1-p)^j$$

with

$$\Pr E^{(t)} = j | A \text{ or } B \text{ is in the argmax.}$$

Given that our coupling to the Polya Urn process is unaffected, we just must show that

$$\Pr E^{(t)} = 0, 1 > \Pr E^{(t)} = 0, 1 | A \text{ or } B \text{ is in the argmax.}$$

The $E^{(t)} = 0$ case is simple since

$$\Pr E^{(t)} = 0 | A \text{ or } B \text{ is the max} = 0$$

since we only use the argmax process when $E^{(t)} \geq 1$ (otherwise we would have assigned the vertex to the least loaded partition). When $E^{(t)} = 1$, this is equivalent to exactly 1 edge being placed and the probability that, of the k bins, it selects an endpoint in A or B is exactly $\frac{m_A^{(t)}+m_B^{(t)}}{t}$. Thus

$$\Pr E^{(t)} = 1 | A \text{ or } B \text{ is the max} =$$
$$\frac{m_A^{(t)}+m_B^{(t)}}{t}\binom{t}{1}p(1-p)^{t-1} \leq$$
$$\binom{t}{1}p(1-p)^{t-1} = \Pr E^{(t)} = 1$$

The result now follows from Theorem 4. □

Proof of Theorem 9: Combining Lemmas 2, 1, and 3, we conclude that Algorithm 3, with k partitions, will asymptotically approach a fractional load of 1 in one partition when run with $p > \frac{2 \log n}{n}$. □

Corollary 2. *Given a single $G(n,p)$ component, for any value $p > \frac{2 \log n}{n}$, Algorithm 1 will eventually concentrate this component into 1 partition as $n \to \infty$.*

This analysis leaves open the question of how long the process must run before one partition dominates the others. This question has been studied by Drinea, Frieze and Mitzenmacher [46]. While they analyze the convergence rates for 2 bins, the proofs can be extended to k bins via the union bound. In the theorem B_0 is the name for one of the two bins and all-but-δ dominant means that B_0 contains at least a $1-\delta$ fraction of the balls thrown. ϵ_0 is the initial amount that the two bins are separated by after n balls and is a constant depending on λ, say $\frac{1}{100\lambda}$.

Theorem 10 (Theorem 2.4 from [46])**.** *Assume that we throw balls into the system until B_0 is all-but-δ dominant for some $\delta > 0$. Then, if $\lambda > 1$, with probability $1 - e^{\Omega(n_0)}$, B_0 is all-but-δ dominant when the system has $2^{x+z} n_0$ balls, where $x = \log_{1+\frac{\lambda-1}{5+4(\lambda-1)}} \frac{0.4}{\epsilon_0}$ and $z = \log_{\frac{2\lambda}{\lambda+1}} \frac{0.1}{\delta}$.*

Lemma 4 extends this theorem to k bins.

Lemma 4 (Lemma 4.1 from [46])**.** *Suppose that when n balls are thrown into a pair of bins, the probability that neither is all-but-δ dominant is upper-bounded by $p(n,\delta)$. Here, we assume $p(n,\delta)$ is non-increasing in n. Then when $1 + kn/2$ balls are thrown into k bins, the probability that none is all-but-γ dominant is at most $\binom{k}{2} p(n,\delta)$ for $\gamma = \delta/(\delta + (1-\delta)/(k-1))$*

To summarize these results on the convergence rate, we find that the attachment process starts in earnest after $\frac{1}{p}$ vertices have arrived. After $\frac{2}{p}$ vertices have arrived, we claim the exponent in the process is greater than 1. From Lemma 4 the probability we do not get an all-but-ϵ domination is inversely polynomial in the number of partitions, $1/\epsilon$ and the number of vertices. The bound given by Theorem 10 holds for $\lambda = 2$ but is loose since λ value increases every after every round of $\frac{1}{p}$ vertices.

Comparisons. From these results, we conclude that the reason that Algorithm 2 fails to concentrate the component is the strict proportionality of its assignments. If instead it used any exponent greater than 1 on its scores, i.e. assign to i proportional to S_i^γ, the concentration result would hold. In particular, there is a huge spectrum of greedy algorithms of the style of arg max Greedy and Proportional Greedy. Amongst these, arg max Greedy provides the strongest possible preference towards concentration.

4.3.4 Extending to $G(\Psi, P)$ graphs and capacity constraints

We showed that with no capacity constraints the arg max Greedy approach is able to asymptotically place a single $G(n,p)$ component into one partition. Specifically, while it will initially place vertices in all partitions, once we begin to see edges, the algorithm concentrates

the component into one partition. By contrasts, the Proportional Greedy approach always cuts the component into k pieces. We would like to extend this analysis for arg max Greedy to graphs that consist of many good clusters but face two challenges - the capacity constraints and the 'bad' inter-cluster edges.

These two challenges motivate our restrictions to both Ψ and P. The capacity constraint can be violated if clusters are of size c and the capacity is $\mathcal{C}$ and more than $\frac{\mathcal{C}}{c}$ communities chose a specific bin to form their large component. From the traditional analysis of throwing m balls into n bins, we know that the expected maximum load (with high probability) is $\frac{\log n}{\log \log n}$ when $m = n$ and $O(\frac{m}{n})$ when $m > n \log n$ [119]. If we can argue that for each cluster the location of its large component is chosen uniformly at random from the bins, then we can use the balls and bins maximum load analysis to argue that if each cluster is small enough, the slack required, $C = (1 + \epsilon)\frac{n}{k}$, is also small. We also require a small amount of slack in the capacities to account for initial mistakes. These mistakes are the result of not seeing edges at the beginning of the process.

For simplicity, our proof will proceed by first assuming that all of the clusters, C_i, are of the same size and that q, the probability of inter-cluster edges, is 0. This will allow us to deal with running l finite Polya Urn processes simultaneously and independently. After this, we show a non-zero bound on q that will bound the probability of the process failing to find a cut on the inter-cluster edges small. Finally, the assumption that the C_i are of equal size can be relaxed by adjusting the parameters in P appropriately.

Lemma 5. *Given a $G(\Psi, P)$ graph where $P_{i,j} = 0$, and $\forall i, |P_{i,i}| > 2 \log n / |C_i|$, let $x_j^{(i)(t)}$ be the fraction of C_i that partition j holds at time t. With no capacity constraints, Theorem 9 will guarantee that, as n grows, for each cluster i, if $\lim_{t\to\infty} x_j^{(i)(t)} = X_j^{(i)}$, then for some j, $X_j^{(i)} = 1$ while all others are 0.*

Proof. This follows directly from Theorem 9 and the fact that when $P_{i,j} = 0$, the individual components can not interact with one another. □

Next, we relax the constraint that there are no edges between components to obtain a bound that still does not necessarily respect capacity constraints.

Lemma 6. *Given a $G(\Psi, P)$ graph with $P_{i,i} = p$, $P_{i,j} = q$, and all l clusters of equal size $|C_i| = |C_j|$ and $p > \frac{2 \log n}{|C_i|}$. Let $x_j^{(i)(t)}$ be the fraction of C_i that partition j holds at time t. With no capacity constraints and k partitions, if $p > 3(k + \sqrt{k} + 1)lq$ then for each cluster i, if $\lim_{t\to\infty} x_j^{(i)(t)} = X_j^{(i)}$, then for some j, $X_j^{(i)} = 1$ while all others are 0.*

Our goal is to bound the number of 'bad' inter-cluster edges away from the number of 'good' intra-cluster edges. We assume worst case distributions so these bounds can safely be relaxed in practice.

Consider component C_i. A natural condition is that there are more expected intra-cluster edges than inter-cluster so $p|C_i| > q(n - |C_i|)$. We require a few more properties. The first is that the inequality holds with reasonable probability so $p|C_i| - \sqrt{p|C_i|} > q(n - |C_i|) +$

$\sqrt{q(n-|C_i|)}$. The second is that we maintain the separation at every step of the execution of the process so $p|C_i|\frac{t}{n} - \sqrt{p|C_i|\frac{t}{n}} > q(n-|C_i|)\frac{t}{n} + \sqrt{q(n-|C_i|)\frac{t}{n}}$. Finally, we also need that the total number of bad edges should be no more than the arg max of the good edges as this guarantees that the bad edges will not affect the concentration results for each component. This adds a factor of k to the bound so we must always guarantee there are at least k 'good' edges for each 'bad' edge.

Proof of Lemma 6: Let the edges from a vertex to its own component be 'good' edges and its external edges be 'bad'. The separation between the good edges and bad edges can be achieved through the use of Chernoff bounds. In particular, at time t, we expect that $|C_i|\frac{t}{n}$ vertices in C_i will have arrived already. Using a Chernoff bound to justify using the expectation, we claim that with probability at least $1-\delta$. Let the next vertex, v, be from C_i . Let $E_i^{(t)}$ be the total number of edges from v to the C_i vertices that have already arrived.

$$E_i^{(t)} > p|C_i|\frac{t}{n} - \sqrt{\log(1/\delta)p|C_i|\frac{t}{n}}.$$

The bad edges, B^t, are drawn from $B(q,(n-|C_i|)\frac{t}{n})$. For clarity, we approximate $n-|C_i|$ as $l|C_i|$. Again, with probability at least $1-\delta$, we claim that

$$B^t < ql|C_i|\frac{t}{n} + \sqrt{\log(1/\delta)ql|C_i|\frac{t}{n}}.$$

We set $\delta = 1/e$ to obtain constant probability at least $1/2$. This assumption is supported by the experimental results in the next Section. We include bounds that hold with high probability in the Appendix.

To add the constraint that the bad edges are less than the $\arg\max\{E_i^{(t)}(j)\}$, we note that the worst case is that all of the bad edges connect to one partition. This can happen if the rest of the graph may not be evenly distributed over the partitions, or we are observing a deviation in the distribution of bad edges. Given this it is sufficient that the number of bad edges is bounded away from the average number of good edges, so we use the condition that

$$p|C_i|\frac{t}{n} - \sqrt{p|C_i|\frac{t}{n}} > k[ql|C_i|\frac{t}{n} + \sqrt{ql|C_i|\frac{t}{n}}]$$

To extract meaningful restrictions on p and q from this equation, we note that $p|C_i|\frac{t}{n} - \sqrt{p|C_i|\frac{t}{n}} > k$ when $t > \frac{(k+\sqrt{k}+1)n}{p|C_i|}$. Similarly, $ql|C_i|\frac{t}{n} + \sqrt{ql|C_i|\frac{t}{n}} < 1$ when $t < \frac{(1/2(3-\sqrt{5})n}{ql|C_i|}$. We find that $\frac{(k+\sqrt{k}+1)n}{p|C_i|} < \frac{(1/2(3-\sqrt{5})n}{ql|C_i|}$ exactly when $p > (k+\sqrt{k}+1)lq/(\frac{1}{2}(3-\sqrt{5}))$. Simplifying, $p > 3(k+\sqrt{(}k)+1)lq$ is sufficient. The gap between the left and right hand sides is monotonically increasing after this point, guaranteeing that all decisions will be made correctly with constant probability. □

Provided $k > 2$, $k + \sqrt{k} + 1 < 2k$ so this bound is more simply $p > 3 * 2klq = 6klq$. If we make stronger assumptions about the distribution of the vertices within the bins at any finite time, i.e. that they are approximately balanced, then we can drop the $(k + \sqrt{k} + 1)$ factor and obtain that $p > 3lq$ is sufficient.

The remaining technical point is the capacity constraints. Given that no aspect of the algorithm is dedicated towards load balancing when edges exist, our only hope can be that the components concentration points are distributed uniformly over the partitions. If this is the case then a standard balls-and-bins analysis will tell us how many components are assigned to each partition. In particular, if n balls are thrown into n bins, we expect the max load to be $\log n$ balls. However, if $n \log n$ balls are thrown into n bins, we expect the max load to be $O(\log n)$. With more than $n \log n$ balls, the maximum load approaches the average.

This approach requires that we be able to argue that the inter-component edges have no affect on the concentration location for each component. This is clear when $q = 0$ and there are no 'bad' edges, the location of the concentration of each component is uniform because of the random ordering of the stream. Similarly, if $p = 1$ then the component is located exactly where the first vertex in the component is placed.

For other values of p and q we must use a more sophisticated argument. In particular, we can exploit the gap between p and q to argue that many intra-component edges are seen before any inter-component edges. If the process has run long enough that we can use Lemma 4 to argue that for each component, one partition contains a bit more than half of the vertices that have arrived, then we can argue that the arg max is never changed by the presence of 'bad' edges and that the processes do not affect each other.

Lemma 7. *Given a $G(\Psi, P)$ graph with $P_{i,i} = p$, $P_{i,j} = q$ satisfying both Lemma 6 and $q = O((k^{2.4} \log l)^{-1})$, with the number of clusters $l > k \log k$ and all clusters of equal size $|C_i|$, with high probability the maximum load of the partitions is bounded by $(1 + \epsilon)\frac{n}{k}$, where ϵ is a function of p, l and k.*

Proof. We first establish that the locations of the concentration for each component is uniformly distributed. This can be done by arguing that the partition that contains the maximum for each component is all-but-δ dominant and applying Theorem 10 and Lemma 4 to obtain that $q = O((k^{2.4} \log l)^{-1})$. The exact calculation is included in the Appendix.

Given the components are uniformly distributed over the partitions, this is a 'balls-and-bins' process with l balls and k bins. If $l = ck \log k$ then with high probability the maximum load is $d_c \log k$ where d_c is a constant depending on c [119]. When $l >> k \log k$ with high probability the maximum load is at most $\frac{l}{k} + \sqrt{2\frac{l}{k} \log k}$. From these results, we conclude that the clusters will be nearly evenly distributed amongst the bins.

Finally, ϵ needs to be set so that the capacity constraints will not be violated by either of the two sources. The first is the distribution of vertices before any edges appear. This is in expectation $\frac{1}{p}$ vertices, and each partition will hold $\frac{1}{pk}$ of them. The other source of slack

required is the exact maximum load. This constant depends on l's relationship to k and can be obtained from [119]. □

The number of vertices required for the above argument to always hold is quite high in the analysis.

Contrast to McSherry's Bounds The same paper that introduces the $G(\Psi, P)$ model also gives algorithms for recovering the cuts in these graphs using spectral methods [103]. In particular, the paper gives bounds for when the spectral algorithm can recover an instance of a planted multisection, a generalization of the balanced partitioning problem we consider. For p and q used as in this paper, if $\frac{p-q}{p} > c\sqrt{\frac{\log n/\delta}{pn}}$, for some large enough constant c, then with probability $1-\delta$, the cut can be recovered. This can be most easily compared with our bound when written as $p-q > c\sqrt{p}\sqrt{\log n/\delta}$ where it becomes clear it is an additive bound, whereas ours is multiplicative. However, the spectral algorithm has access to significantly more information so an improved bound is to be expected.

4.4 Experimental Evaluation

The proofs in the previous sections show that for a certain range of parameters and size graph, the algorithm succeeds in recovering a good partitioning. It leaves open some interesting questions that can be experimentally evaluated:

- What is the relationship between ϵ, the load balancing factor in Lemma 7 and k, the number of partitions, and l, the number of components?
- How tight are the bounds? It is necessary that the density of edges within components is $p > \frac{2\log n}{|C_i|}$ or that the gap between p and q, the probability of edges between components is at least $p > 6klq$?
- Are the convergence rates tight? For what size graph do we begin to recover the partitioning?
- When we are asymptotically recovering the partitioning, can we quantify how many mistakes we are making, i.e. how many vertices are separated from their components at the end of the process?

These questions are ideals candidate for experimental simulation. In fact, experimental results here can lead to a much better understanding of the algorithm than theoretical worst case bounds. In the following, using values that satisfy Lemma 6, we generate $G(\Psi, P)$ graphs and see how well arg max Greedy recovers the embedded cut.

Evaluation. Given a setting of the parameters, we generate a random $G(\Psi, P)$ graph and run the algorithm 25 times, each with a different random ordering. After each run, for

each component in the $G(\Psi, P)$ graph, we its largest part in the partitioning i.e. if C_i is the component, and $P_1, P_2, \cdots P_k$ the final partitioning, we calculate $\max_{j \in k} |C_i \cap P_j|/|C_i|$. The theorems predicts that for all components, this value approaches 1 as the graph grows. Note that it can never be worse than $\frac{1}{k}$ for k partitions.

4.4.1 Load Balancing Factor

Understanding the load balancing factor required is the first step to understanding the other constraints. This is because if the load balancing factor is set too low, we will see this in the error calculations. To understand the slack required, we explore two settings of p and q, $p = 1$ and $q = 0$ or $q = \frac{p}{6kl}$ where l, the number of components is larger than $k \log k$. Now, for each size graph, we run the algorithm 20 times and record the number of partitions that hit their capacity constraints. We also vary l to understand how its relationship with k affects the required slack. We include 3 figures to demonstrate the relationship. The first,

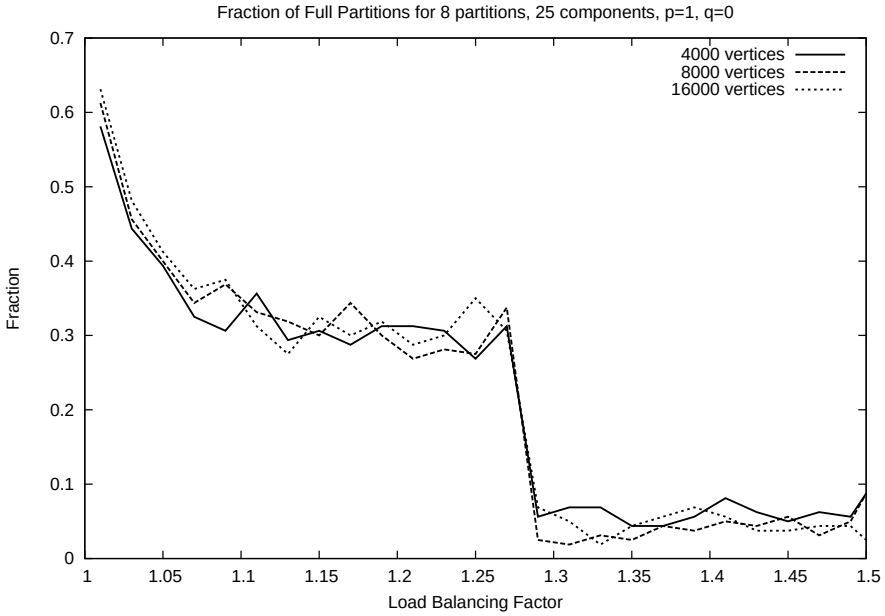

Figure 4.1: Load balancing is not a function of the size of the graph

Figure 4.4.1 shows the fraction of full partitions when ϵ is allowed to range from 0.01 to 0.5 for graphs of size 4,000, 8,000 and 16,000. There is no difference between the threshold point in these graphs. The second, Figure 4.2 shows that fixing p, q and k but increasing l, the number of components, yields significantly better load balancing factors. The third Figure 4.3 shows that whether $q = 0$ or $q = 0.002 = p/6kl$, the load balancing appears the same.

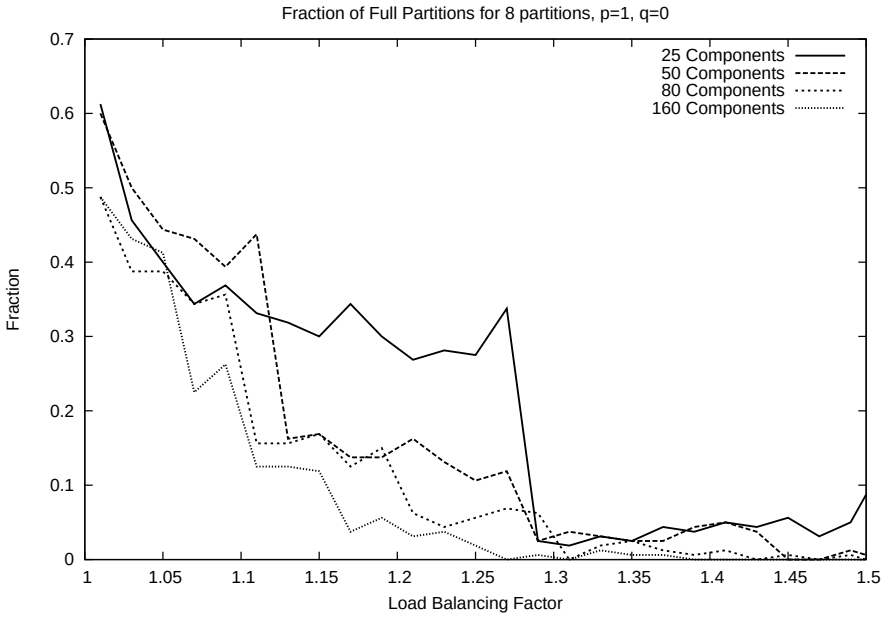

Figure 4.2: Increasing the number of components improves the load balancing.

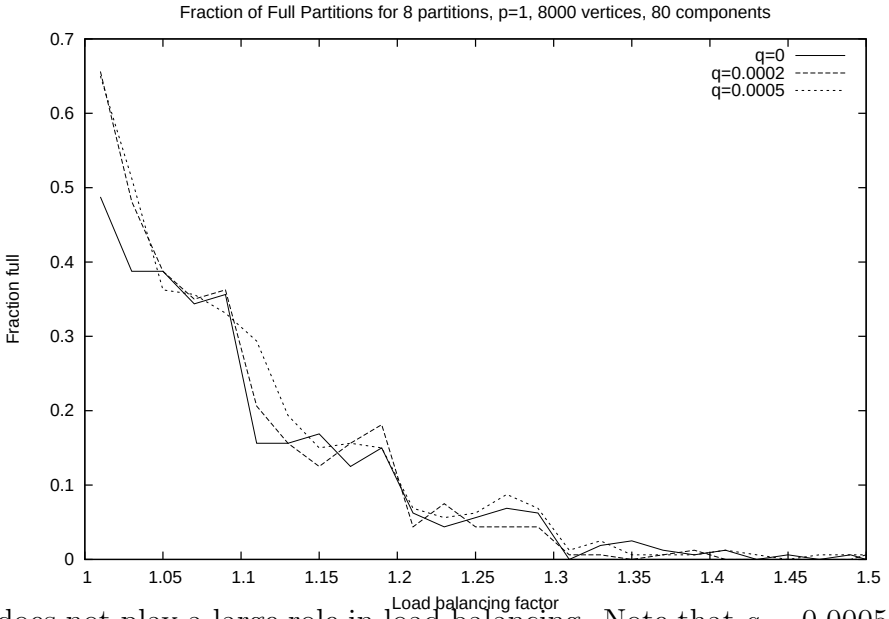

Figure 4.3: q does not play a large role in load balancing. Note that $q = 0.0005$ is above the threshold required by the Theorems.

4.4.2 Density Requirement

Lemma 6 requires that each component have edge density at least $p > \frac{2 \log n}{|C_i|}$. To explore whether this is necessary, we can fix values for q, k and l and let p range above and below $\frac{2 \log n}{|C_i|}$. For each run, we measure the error from the perfect solution by looking at the Euclidean distance between the length-l vector of the values of $\max_{j \in k} |C_i \cap P_j|/|C_i|$ and the all-ones vector.

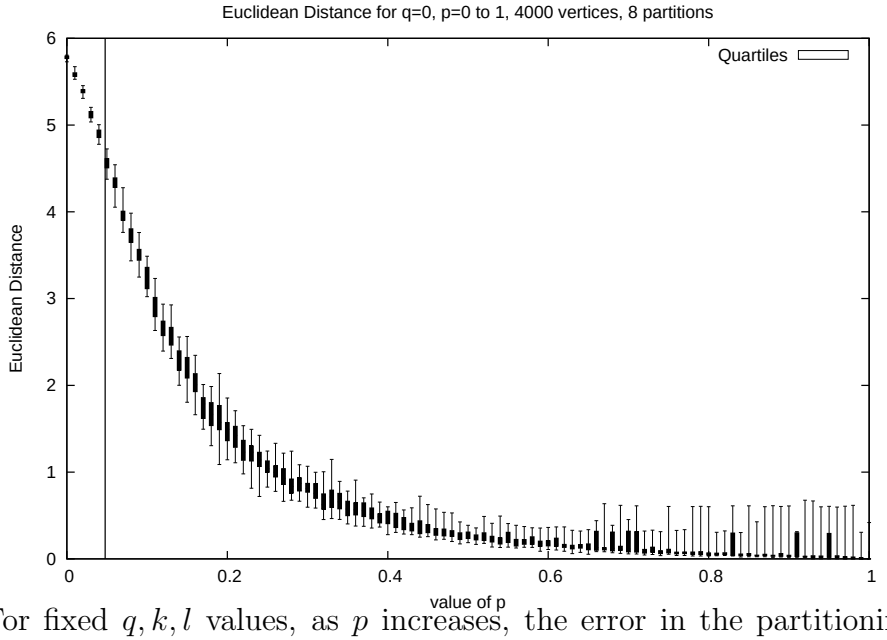

Figure 4.4: For fixed q, k, l values, as p increases, the error in the partitioning generated drops to 0. The vertical bar marks the value required by the theorems.

Though not pictured in Figure 4.4, the graph size shifts the 'elbow' of the graph to the left with a sharper transition, matching the bound of the theorem.

4.4.3 Constraints on q

As in the experiments to understand the density factor, we can also fix values for p, k and l and let q range above and below $\frac{p}{6kl}$. Is the factor of k necessary? We measure the error by Euclidean distance as above.

We clearly see the effect that increasing q has on the algorithm's ability to recover the partitioning in Figure 4.5. While the value required by the theorems seems unnecessarily small (and can only be seen by zooming in on this page), dropping the required factor of k and using $q = 0.02$ obtains an average error of only 0.07 over 25 runs when the maximum error is 7.

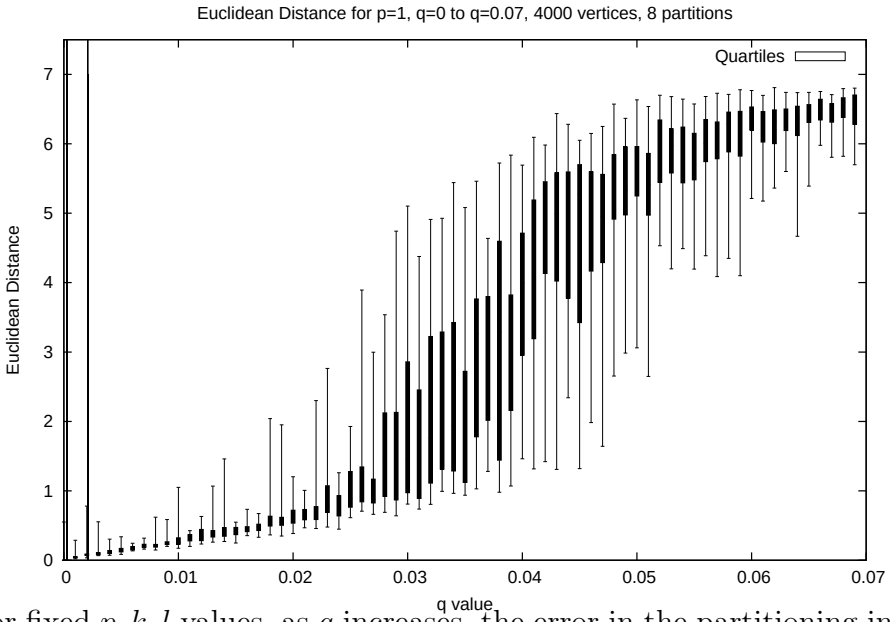

Figure 4.5: For fixed p, k, l values, as q increases, the error in the partitioning increases from 0 to maximum error. The leftmost vertical bar (at 0.00026) marks the value required by the theorems, while the second (at 0.0021) is $q = p/6l$.

4.4.4 Convergence Rate

The values given by the Theorems in [46] about the rate of convergence imply a somewhat pessimistic bound - $q = O((k^{2.4} \log l)^{-1})$. We can evaluate this bound by fixing p, q, k and l and letting the size of the graph grow. As it grows, we can measure the Euclidean distance to find how quickly it is able to obtain good results in terms of recovering the partitioning.

The settings for the algorithm in Figure 4.6 were $p = 0.75, q = \frac{p}{6kl}, k = 8, l = 100$. The graph size range from 400 to 51,200 vertices. We see that as the size of the graph increases, the euclidean distance from the optimal partitioning solution quickly drops. For 51,200 vertices, the median error for 25 runs is only 0.04. This is despite the fact that the theorem required that $q < 0.000013$ whereas we used $q = 0.00015625$.

4.5 Conclusions and Future Work

We have studied two simple greedy algorithms for streaming balanced graph partitioning. We first showed lower bounds on the possible approximation ratio obtainable by any algorithm and then analyzed two variants of a randomized greedy algorithm on a random graph model with embedded balanced k-cuts. On these graphs we were able to explain previous experimental results showing that the arg max Greedy algorithm is able to recover a good partitioning while the Proportional Greedy variant is not. Our proof connects the greedy

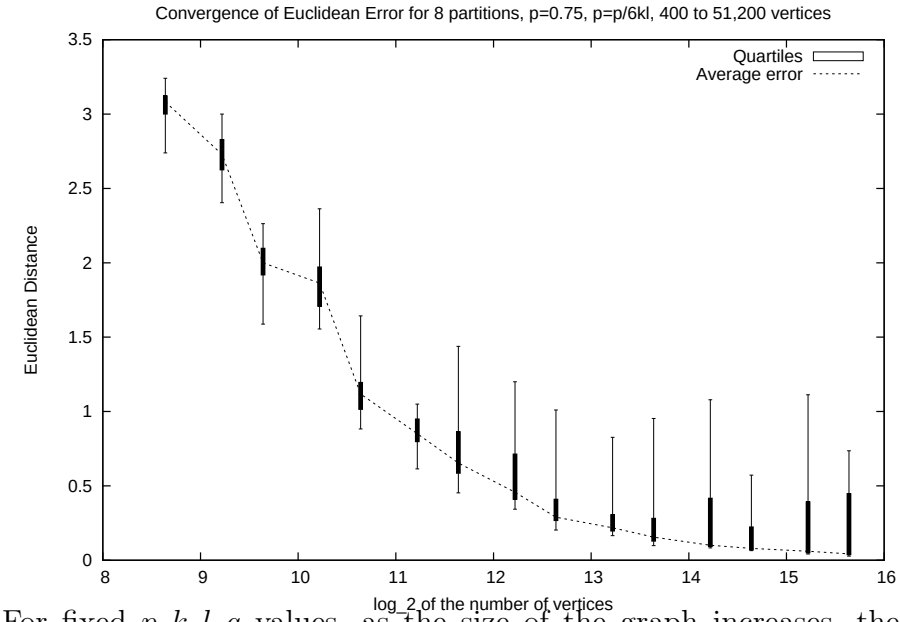

Figure 4.6: For fixed p, k, l, q values, as the size of the graph increases, the error in the partitioning generated drops to 0.

algorithms with finite Polya urn processes and exploits concentration results about those processes.

There are several interesting future directions. The first is to improve the parameters of the analysis presented in this paper. The experiments show that the deterministic greedy algorithm continues to work with larger amounts of noise than that allowed by our theorems. Also, our analysis depends heavily on the assumption of independence between the vertices and the edges. Again, the experiments in [136] show that the deterministic greedy algorithm performs well on other random graph models, like Watts-Strogatz, although it may not obtain a provably good result there. Explaining this and proving results about the approximation ratio is an interesting question.

Perhaps most important future direction for other streaming algorithms is that in [136], additional stream orderings were studied, namely a breadth-first search ordering, and a depth-first search ordering. Experimentally, the algorithms tested all performed better on both of these orderings than the random ordering. An interesting open question is to develop techniques for analyzing streaming graph algorithms on these orders, and whether lower bounds can be developed.

4.6 Appendix

4.6.1 High Probability Bounds for Lemma 6

The experiments justify the assumption that we only need the following two statements to hold with constant probability:

$$E_i^{(t)} > p|C_i|\frac{t}{n} - \sqrt{p|C_i|\frac{t}{n}}$$

$$B^t < ql|C_i|\frac{t}{n} + \sqrt{ql|C_i|\frac{t}{n}}$$

Requiring each to hold with probability $1-\delta$ increases the gap required from $p > 3(k + \sqrt{k}+1)kql$ by adding a dependency on δ. In particular, redoing the calculations, we have that

$$p|C_i|\frac{t}{n} - \sqrt{\log(1/\delta)p|C_i|\frac{t}{n}} > k$$

exactly when

$$t > \frac{n}{p|C_i|}(k + \log(1/\delta)/2 + \sqrt{k\log(1/\delta) + (\log(1/\delta)^2/4)})$$

Similarly,

$$ql|C_i|\frac{t}{n} + \sqrt{\log(1/\delta)ql|C_i|\frac{t}{n}} < 1$$

exactly when

$$t < \frac{n}{ql|C_i|}(1 + \log(1/\delta)/2 - \sqrt{\log(1/\delta) + (\log(1/\delta)^2/4)})$$

Solving these two equations as in Lemma 7 gives us a similar relationship that $p > f(\delta)kql$.

4.6.2 Calculation of q for Lemma 7

In order to prove Lemma 7 we need to understand for a given setting of p and q how much interaction between the components there is at the t^{th} vertex. In particular, for the t^{th} vertex, we expect that there will be $p\frac{t}{l}$ edges from that vertex to its own component (good edges) and $q\frac{(l-1)t}{l}$ edges to other components (bad edges). Provided $t < \frac{l}{q(l-1)}$, we do not expect any bad edges so the components do not interact at all.

When we do begin to see bad edges, we can appeal to Lemma 4. If it is the case that for the given component, one partition contains a $1/2 + x$ fraction of the component that

has arrived to this point, and all other partitions split the remaining $1/2 - x$ fraction then we can argue that the bad edges do not affect the concentration of the process provided the arg max for the good edges is not changed by the addition of the bad edges. Specifically, we are concerned with $t = \frac{l}{q(l-1)} > \frac{1}{q}$ so we can find x by solving:

$$(1/2 + x)p\frac{t}{l} - \sqrt{(1/2 + x)p\frac{t}{l}} > ((1/2 - x)p\frac{t}{l} + \sqrt{(1/2 - x)p\frac{t}{l}}$$

The above equation gives the distribution of the good edges at time t. Substituting that $t = \frac{1}{q}$ and there is only one bad edge, we need that

$$(1/2 + x)\frac{p}{ql} - \sqrt{(1/2 + x)\frac{p}{ql}} > ((1/2 - x)\frac{p}{ql} + \sqrt{(1/2 - x)\frac{p}{ql}}$$

This results in

$$x = \pm\sqrt{\frac{2(p/ql)^3 - (p/ql)^4}{4(p/ql)^4}}$$

From this, we can gather that a sufficient γ value required for Lemma 4 is $\gamma = \frac{1}{2} - \sqrt{1/2(p/ql)}$. Lemma 4 gives a formula for translating this γ into a δ value for Theorem 10. Solving for δ we get that

$$\delta = \frac{\gamma}{k - 1 - (k - 2)\gamma}.$$

Plugging in our γ value, we obtain that

$$\delta = \frac{1/2 - \sqrt{1/2(p/ql)}}{k - 1 - (k - 2)(1/2 - \sqrt{1/2(p/ql)})}.$$

We can simplify this by claiming that $\delta < \frac{1}{k}$ is sufficient.

The failure probability that we need to obtain from Theorem 10 for Lemma 4 is at most $\frac{c}{k^2 l}$ to use a union bound and still obtain a constant probability of success for the whole process. Therefore, we need to set $n_0' = n_0 + 2\log k + \log l$.

From here, we can obtain a number of balls thrown before we can obtain this level of concentration. In particular, we need $2^{x+z}n_0$ balls, where $x = \log_{1+\frac{\lambda-1}{5+4(\lambda-1)}} \frac{0.4}{\epsilon_0}$ and $z = \log_{\frac{2\lambda}{\lambda+1}} \frac{0.1}{\delta}$. The x term allows us to obtain up to all-but-0.1 dominance, while the second improves the result to all-but-δ dominance. Therefore, if $k \leq 10$, then we only need $n_0 2^x$ balls. More generally, substituting that $\epsilon_0 = 1/5\lambda$ and $\delta = \frac{1}{k}$, this value becomes:

$$(2\lambda)^{1/\log_2(5\lambda/(1+4\lambda))}(0.1k)^{1/\log_2(2\lambda/(\lambda+1))}n_0'$$

The interesting thing about the process is that as more vertices arrive, the λ value increases. From this, we can immediately claim that this equation dramatically over-estimates the number of vertices needed before 2 bins would obtain a state with all-but-$\frac{1}{k}$ dominance.

In particular, for the p and q values required by Lemma 6, we have $p = 6klq$ so λ reaches a value of $3k$ before we expect to see bad edges. Unfortunately, the best we can assume is that $\lambda = 2$ obtaining the following value:

$$4^6.578(0.1k)^2.4n_0' \approx 9127n_0'(0.1k)^{2.4}$$

It is certainly possible to set q to $1/9127n_0'(0.1k)^{2.4}$ but it is a significantly different bound from $p > 6klq$.

Part II

The Use of Matchings in Solving Graph Problems

Chapter 5

An Introduction to Matching Algorithms

5.1 Introduction

A matching in a graph is one of the most basic concepts in graph theory. Simply put, a matching is a set of edges such that no vertex is matched more than once. More formally, given a graph $G = (V, E)$, a matching M is a set of edges E such that for all $v \in V$, the set of edges of M that contain v has size at most 1. If all vertices are matched, this is called a *perfect matching*. If no more edges can be added to M without violating the constraint, this is called a *maximal matching*.

For clarity, consider Figure 5.1. A simple graph is given with 6 vertices and 7 edges. The left most graph has no matching marked. The middle graph's red edges form a maximal matching since no other edge can be colored red without neighboring an already red edge. However, the right most graph has the edges of a perfect matching colored blue. Every vertex is a member of exactly one blue edge.

Where are matchings used? Matchings form a fundamental algorithmic building block for graph algorithms. Aside from the uses we will see in the following chapters, they are used in a vast array of applications from matching medical residents with hospitals to data layout for optimizing communication in distributed systems.

The wide spread usage of matchings has lead to the study of a seemingly endless number of variants. Let M be the set of edges of G included in the matching.

Maximum cardinality: The goal of this version of the problem is to maximize the size of M subject to the constraints.

Maximum weight: The goal of this version is not to maximize the size of M but to maximize the sum of the weights of the edges included in M. If the graph is unweighted, this is equivalent to maximum cardinality.

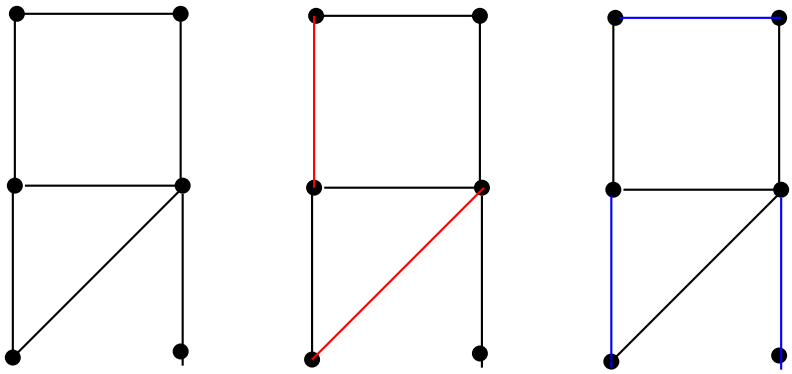

Figure 5.1: Three copies of the same graph. The middle graph with the red edges represents a maximal matching - no other edge can be color red without neighboring an already red edge. The right most graph is a perfect (and maximum) matching.

Minimum weight maximum cardinality: The goal of this version is to find a set M of maximal size that has the smallest sum of edge weights possible. Without the constraint that M be of a certain size this version is trivially solved by the empty set.

Bipartite versions: The above problems are often significantly simplified if we can assume the graph is bipartite.

b-matchings: It is often the case that we don't want to have a strict one-to-one matching. For example, if we are matching bid slots to advertisers we may wish to allow each advertiser to be matched to multiple ads. This is a generalization of standard matching and all of the above variants can apply to b-matching.

Randomly generated: It is often the case that there are many matchings the satisfy a given variant. For example, a complete bipartite graph with n vertices on each side has $n!$ distinct perfect matchings. It is often the case that we would like to select a random matching satisfying the given goal. This variant will be used in depth in Chapter 6.

Stable matchings: The classic stable marriage problem is an example of a matching problem. Given two sets of vertices, U and V, with each vertex in U having a ranking over V (and vice versa), the goal is to match the vertices of U to V so that no pair of matched vertices (u_1, v_1) and (u_2, v_2) prefers the other to their partners, i.e. u_1 ranks v_2 above v_1 and v_2 ranks u_1 above u_2.

There also exist a multitude of algorithms for solving the matching problem, from approximation algorithms for max-weight matchings to parallel, distributed, and streaming algorithms for the traditional maximum cardinality problem. We will briefly outline some of the approaches now.

One of the most interesting complexity results about matchings is that finding a single maximum matching can be solved very efficiently in polynomial time. However, counting how

many distinct maximum matchings a graph has is equivalent to computing the permanent of its adjacency matrix. This problem one of the earliest #P-Complete problems.

5.2 Classic Matching Algorithms

There are many different styles of algorithms for solving the problem of finding a matching in a graph. These are usually separated by whether they are intended for bipartite graphs or general graphs. We will begin by reviewing the classical approaches for finding matchings sequentially and then move to newer algorithms for finding matchings using parallel and distributed algorithms.

5.2.1 Integer and Linear Programming

The use of various forms of convex programming has been a boon to algorithm design ever since Dantzig designed the simplex algorithm. The matching problem has an exceedingly simple polytope. We have m variables, one for each edge, x_e and n constraints, one for each vertex.

The simplest integer program is that for maximum cardinality matching and is:

$$\begin{aligned} & \max \textstyle\sum_e X_e \\ \text{subject to} \quad & \forall v \in V, \textstyle\sum_{e \ni v} X_e \leq 1 \\ & \forall e, X_e \in \{0,1\} \end{aligned}$$

If we wished to instead solve maximum weight matching, we could simply change this integer program to include the weights on the edges, w_e:

$$\begin{aligned} & \max \textstyle\sum_e w_e X_e \\ \text{subject to} \quad & \forall v \in V, \textstyle\sum_{e \ni v} X_e \leq 1 \\ & \forall e, X_e \in \{0,1\} \end{aligned}$$

Given that solving integer programs is NP-hard, the standard approach is to relax the integer constraints and obtain a linear program. In this case, we find that the linear program for maximum cardinality matching is

$$\begin{aligned} & \max \textstyle\sum_e X_e \\ \text{subject to} \quad & \forall v \in V, \textstyle\sum_{e \ni v} X_e \leq 1 \\ & \forall e, X_e \geq 0 \end{aligned}$$

We can now solve this program using any LP solving algorithm. The interior point algorithm guarantees that this can be solved in time polynomial in the number of constraints. The general approach at this point would then involve trying to round a fractional solution

into an integer solution and obtaining an approximation algorithm. However, this particular linear program consists of a unimodular matrix meaning that its inverse consists only of integers. Because of this, when we find a vector of assignments that maximizes $\sum_e X_e$, it will be the case that despite relaxing the integer constraints, all entries are either 0 or 1. This shows that maximum cardinality matching can be solved exactly in polynomial time.

Maximum weight matching In 1965, Edmonds also showed that maximum weight matchings can be found in polynomial time [49] precisely because the corners of the matching polyhedra occur at integer points. The original implementation of this algorithm required $O(n^3)$ time. Others worked on improving the running time of this algorithm until Tarjan and Gabow achieved an $O(m\sqrt{n})$ algorithm [59]. Relying on a matrix multiplication approach, Sankowski was able to remove the dependence on m, the number of edges, and gave an $O(Nn^\omega)$ algorithm, where N is the maximum edge weight and n^ω is matrix multiplication time, currently at $\omega < 2.3727$ [150]. The next approach is to investigate approximation algorithms with the goal of developing a near linear in m algorithm. Duan and Pettie do so and give a $(1-\epsilon)$ approximation algorithm in $O(m\epsilon^{-2}\log^3 n)$ time [47].

5.2.2 Bipartite Flow Gadget

When $G = (U, V, E)$ is a bipartite graph, the question of finding a matching takes on a much more obvious meaning, with examples including the stable marriage problem. Additionally, from an algorithmic perspective, it admits a new approach to the problem by exploiting a connection with the maximum flow problem.

The *maximum flow problem* for a graph has many variants. The simplest is known as single-source, single-sink flow. We are given a directed graph with edge capacities. Two vertices are labeled the source s, and the sink t. The goal is to route as much flow as possible from s to t without violating the capacity constraints. As with matching, this is a classic graph problem in computer science and a vast literature exists studying it and variants. However, the first algorithm to solve this problem in polynomial time was developed in 1954 by Ford and Fulkerson in $O(Ef)$ time [58]. Various improvements have been developed over the years, including the Edmonds-Karp algorithm [51], Dinitz's blocking flow algorithm, Goldberg and Tarjan's push-relabel algorithm [66] and, the current fastest, Goldberg-Rao [65].

The gadget construct consists of adding a source s and a sink t to the graph. The sink s is connected with all of the vertices in U and the edges are given capacity 1. All of the edges in E are directed from U to V, and all of the vertices in V are connected with the sink t with capacity 1. We can now use any maximum flow algorithm and the edges used by the flow will exactly correspond to a maximum cardinality matching.

5.2.3 Augmenting Paths and Edmonds Algorithm

We can also design algorithms specifically for solving the maximum cardinality matching problem without using machinery designed for other problems. To do so, we must first introduce the idea of an *augmenting path*. In the context of a graph $G = (V, E)$ and a current matching M, an augmenting path P consists of edges $(v_1, v_2), (v_2, v_3), \ldots (v_{k-1}, v_k)$ where the edges alternately belong to M and $E \setminus M$. If it is the case that v_1 and v_k are not currently matched by M, then we can augment M to M' by $M' = M \setminus P \cup P \setminus M$. It remains to discuss how we can find augmenting paths and how can we do so efficiently.

Bipartite graphs When $G = (U, V, E)$ is bipartite, finding augmenting paths becomes straightforward. We begin by making all edges directed from the vertices in U to the vertices in V. As we build M, we change the direction of the edges in M to point from V to U. Call a vertex 'free' if it is not included in M right now. The problem of finding an augmenting path is now exactly the same a connectivity problem of finding a path from the free vertices of U to the free vertices of V. Naively, this problem can be solved using nothing more sophisticated than breadth-first search in time $O(VE)$. This running time is improved by the Hopcroft-Karp algorithm to $O(\sqrt{V}E)$ by utilizing each breadth-first search path to match more than 1 free vertex if possible.

Non-bipartite graphs For non-bipartite graphs, we run into the problem that we can't use breadth-first search because we don't necessarily know which end of the path a vertex belongs on. Specifically, we can have situations where we have an augmenting cycle of odd length so that going around it one way can result in an edge from M being used, while using the other direction will result in an edge from $E \setminus M$ being needed to continue the path.

This problem was first solved by Edmonds' algorithm [50], discovered in 1961. Edmonds' called these augmenting cycles *blossoms* and the key insight of the algorithm is that these can be contracted in a way that allows us to preserve the leveled structure of the search style algorithm. This algorithm runs in $O(|V|^4)$ time, but variants have been improved to $O(|E||V|^{1/2})$ matching the running time for exact bipartite matching.

5.2.4 Finding Matchings In Parallel

A classic result in randomized algorithms is the Mulmuley, Vazirani, Vazirani PRAM perfect matching algorithm. To explain it, we must first draw connections between the permanent/determinant of a matrix and perfect matchings.

Recall that the definition of the determinant of an $n \times n$ matrix A is

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}.$$

The permanent of the same matrix differs only in that the sign of the permutation is not included.

$$\text{per}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

This subtle difference turns out to have vast implications in computation time. While we can calculate a determinant in polynomial time (at worst $O(n^3)$ with Gaussian elimination), the permanent is a classic #-P problem.

However, if we consider A to be an adjacency matrix with 0 in the entries with no edge and 1 for edges, then the permutations are exactly all possible perfect matchings in the graph. The permanent of the graph is exactly equal to the total number of perfect matchings. We can use this fact to develop matrix based perfect matching algorithms (as well as approximation algorithms for the permanent).

To develop the MVV algorithm, we first must introduce the Tutte matrix. Rather than considering the 0-1 adjacency matrix, if the (i, j) entry of A is 1, replace it with a variable $x_{i,j}$ if $i < j$ and $-x_{i,j}$ if $i > j$. Call this matrix $M(G)$. The first result that we need is:

Lemma 8. *The determinant of $M(G)$ is non-zero if and only if G has a perfect matching.*

The above lemma means that when we take the polynomial that results from $\det(M(G))$, it is identically equal to 0 whenever there is not a perfect matching. This observation connects finding a perfect matching with identity testing and the Schwartz-Zippel test. Given that this is a n degree polynomial, it can have at most n roots if it is non-zero. If we sample a number between 0 and $2n$, the probability that we sample a root is at most $1/2$. This gives us a randomized algorithm for deciding if a polynomial is identically 0.

However, simplifying applying the Schwartz-Zippel test to the determinant of $M(G)$ only gives us a randomized polynomial time sequential algorithm. In order to create a PRAM algorithm, we must introduce the isolating lemma.

Lemma 9. *(Isolating Lemma) Let n and N be positive integers and let F be an arbitrary family of subsets of the universe $\{1, \cdots, n\}$. Suppose each element $x \in \{1, \cdots, n\}$ in the universe receives an integer weight $w(x)$, each chosen uniformly at random from $\{1, \cdots, N\}$. The weight of a subset $S \in F$ is $w(S) = \sum_{x \in S} w(x)$. Then with probability at least $1 - n/N$, there is a unique subset in F with minimum weight.*

How is the above lemma useful? If we assign random values between 1 and N to the variables in $M(G)$ then it states that with probability $1 - n/N$, there exists a unique perfect matching with minimum weight. This will allow us to break ties in parallel (by always preferring the minimum weight) and allow us to find all of the edges in the matching at once.

Specifically for the MVV algorithm, we let N be at least $2n$ so there is probability at least $1/2$ of there being a unique minimum weight perfect matching. One additional trick is that the isolating lemma discusses the sum of the weights while the permanent computes

the products. To handle this, if $w_{i,j}$ is the randomly selected value for the (i, j) edge, then Let B be the matrix $M(G)$ with $X_{i,j}$ replaced by $2^{w_{i,j}}$. Now,

Algorithm 5 The Mulmuley, Vazirani, Vazirani matching algorithm

calculate 2^w, the largest power of w that divides $\det(B)$
for each edge (i, j) in parallel **do**
  compute $t_{i,j} = \det(B_{i,j})\frac{2^{w_{i,j}}}{2^w}$, where $B_{i,j}$ is B with the i and j columns and rows removed
  place (i, j) in the matching M iff $t_{i,j}$ is an odd integer

If there is a unique minimum weight perfect matching, this algorithm will produce it. If one does not exist, we have no guarantees about the output.

5.2.5 Finding Matchings in a Map Reduce Framework

In 2004, the MapReduce computing platform was officially announced [45]. This model for distributed computing differs from previous theoretical models, including Turing Machines, the PRAM model, and the BSP model. As a result, new models of computation on MapReduce systems have been developed.

A MapReduce system consists of some cluster of machines [87]. The data is available on the system in the form of *¡key,value¿* pairs. There are two phases of computation. The first is *map*. During a map phase all of the data is processed on tuple at a time. All pairs with the same *key* value are mapped and sent to the same machine. The second phase *reduce* takes all of the pairs sent to each machine and performs some transformation. Commonly, the data is aggregated or summed. New *¡key,value¿* pairs are produced and the procedure can repeat if desired.

Communication between machines in MapReduce can only happen by exchanging key-value pairs during the map phase. This makes it similar to the PRAM model in terms of communication. The primary constraint applies to the amount of memory available. For a problem of size N, we assume that there are $N^{1-\epsilon}$ machines and no machine has more than $O(N^{1-\epsilon})$ memory available. This means the total memory available in the system is $O(N^{2-2\epsilon})$. In practice, the map phase can be very slow so the goal is to minimize the number of rounds of communication needed.

We present the *filtering* method for finding a maximal matching first. The maximal matching problem is relatively simple as it does not require a great amount of synchronization. Begin by distributing the edges of the graph uniformly at random across the machines. To find a maximal matching, we begin by sampling the edges with the goal of generating a small set of edges that can fit in one machine. We can then find a maximal matching on that sample, remove all of the incident edges from the graph and then repeat. Provided the sample is large enough, you can argue that the graph is reduced in size fast enough that only $O(\log n)$ rounds are required [87].

The above approach can be improved from $O(\log n)$ rounds to 3 when enough memory is available. Specifically, if each machine holds $N^{1-\epsilon}$ memory and η, the expected size of the sample, is $n^{1+2c/3}$ then we only require 3 rounds.

Unfortunately, the above algorithm can not be easily extended to one that finds a maximum cardinality matching. It can be the case that only one augmenting path exists and it uses the entire graph (envision a path graph where the two end points are unmatched). Finding the path requires a global view of the data.

We can, however, extend the algorithm to an approximation algorithm for finding a maximum weighted matching. This problem can be solved exactly with full information, but approximation algorithms are a popular way to improve the running time. The most common technique for this is to let W denote the maximum weight edge in the graph. We can now separate the edges of the graph into classes based on W, i.e. $\log W$ classes of weight between 1 and 2, 2 and 4, 4 and 8, on through $\frac{W}{2}$ and W. From here, we can now use the maximal matching algorithm as a blackbox on each of the classes. Beginning with the $\frac{W}{2}$ to W class, find a maximal matching on just these edges. We could sequentially apply the algorithm, each time removing the edges that had been matched in the previous round. However, assuming enough edges, this would require $3 \log W$ rounds. To fix this, we can find maximal matchings on all of the weight classes simultaneously. We can stitch these together sequentially by considering the heaviest edges first and only adding an edge if it is a valid addition. It can be proved that this method obtains an 8-approximation the the maximum weight matching using 4 MapReduce rounds [87].

Chapter 6

The Joint Degree Distribution

6.1 Introduction

Graphs are widely recognized as the standard modeling language for many complex systems, including physical infrastructure (e.g., Internet, electric power, water, and gas networks), scientific processes (e.g., chemical kinetics, protein interactions, and regulatory networks in biology starting at the gene levels through ecological systems), and relational networks (e.g., citation networks, hyperlinks on the web, and social networks). The broader adoption of the graph models over the last decade, along with the growing importance of associated applications, calls for descriptive and generative models for real networks. What is common among these networks? How do they differ statistically? Can we quantify the differences among these networks? Answering these questions requires understanding the topological properties of these graphs, which have lead to numerous studies on many "real-world" networks from the Internet to social, biological and technological networks [55].

Perhaps the most prominent theme in these studies is the skewed degree distribution; real-world graphs have a few vertices with very high degree and many vertices with small degree. There is some dispute as to the exact distribution, some have called it power-law [22, 55], some log-normal [13, 117, 106, 28], and but all agree that it is 'heavy-tailed' [43, 125]. The ubiquity of this distribution has been a motivator for many different generative models and is often used as a metric for the quality of the model. Models like preferential attachment [22], the copying model [82], the Barabasi hierarchical model [121], forest-fire model, the Kronecker graph model [90], geometric preferential attachment [56] and many more [91, 141, 31] study the expected degree distribution and use the results to argue for the strength of their method. Many of these models also match other observed features, such as small diameter or densification [80].

The degree distribution alone does not define a graph. McKay's estimate [102] shows that there may be exponentially many graphs with the same degree distribution. However, models based on degree distribution are commonly used to compute statistically significant structures in a graph. For example, the modularity metric for community detection in

graphs [109, 108] assumes a null hypothesis for the structure of a graph based on its degree distribution, namely that probability of an edge between vertex v_i and v_j is proportional to $d_i d_j$, where d_i and d_j represent the degrees of vertices v_i and v_j. The modularity of a group of vertices is defined by how much their structure deviates from the null hypothesis, and a higher modularity signifies a better community. The key point is that the null hypothesis is solely based on its degree distribution and therefore might be incorrect. Degree distribution based models are also used to predict graph properties [104, 8, 41, 40, 42].

These studies improve our understanding of the relationship between the degree distribution and the structure of a graph. The shortcomings of these studies give insight into what other features besides the degree distribution would give us a better grasp of a graph's structure. For example, the degree assortativity of a network measure whether nodes attach to other similar or dissimilar vertices. This is not specified by the degree distribution, yet studies have shown that social networks tend to be assortative, while biological and technological networks tend to be dissortative [112, 111]. An example of recent work using assortativity is [130]. In this study, a high assortativity is assumed for connections that generate high clustering coefficients, and this, in addition to preserving the degree distribution, results in very realistic instances of real-world graphs. Another study that has looked at the joint degree distribution is dK-graphs [97]. They propose modeling a graph by looking at the distribution of the structure of all sized k subsets of vertices, where $d = 1$ are vertex degrees, $d = 2$ are edge degrees (the joint degree distribution), $d = 3$ is the degree distribution of triangles and wedges, and so on. It is an interesting idea, as clearly the nK distribution contains all information about the graph, but it is far too detailed as a model. At what d value does the additional information become less useful?

One way to enhance the results based on degree distribution is to use a more restrictive feature such as the *joint degree distribution*. Intuitively, if degree distribution of a graph describes the probability that a vertex selected uniformly at random will be of degree k then its joint degree distribution describes the probability that a randomly selected *edge* will be between nodes of degree k and l. We will use a slightly different concept, the joint degree matrix, where the total number of nodes and edges is specified, and the numbers of edges between each set of degrees is counted. Note that while the joint degree distribution uniquely defines the degree distribution of a graph up to isolated nodes, graphs with the same degree distribution may have very different joint degree distributions. We are not proposing that the joint degree distribution be used as a stand alone descriptive model for generating networks. We believe that understanding the relationship between the joint degree distribution and the network structure is important, and that having the capability to generate random instances of graphs with the same joint degree distribution will help enable this goal. Experiments on real data are valuable, but also drawing conclusions only based on a limited data may be misleading, as the graphs may all be biased the same way. For a more rigorous study, we need a sampling algorithm that can generate random instances in a reasonable time, which is the motivation of this work.

The primary questions investigated by this chapter are: Given a joint degree distribution and an integer n, does the joint degree distribution correspond to a real labeled graph? If

so, can one construct a graph of size n with that joint degree distribution? Is it possible to construct or generate a *uniformly random* graph with that same joint degree distribution? We address these problems from both a theoretical and from an empirical perspective. In particular, being able to uniformly sample graphs allows one to empirically evaluate which other graph features, like diameter, or eigenvalues, are correlated with the joint degree distribution.

Contributions We make several contributions to this problem, both theoretically and experimentally. First, we discuss the necessary and sufficient conditions for a given joint degree vector to be graphical. We prove that these conditions are sufficient by providing a new constructive algorithm. Next, we introduce a new configuration model for the joint degree matrix problem which is a natural extension of the configuration model for the degree sequence problem. Finally, using this configuration model, we develop Markov Chains for sampling both pseudographs and simple graphs with a fixed joint degree matrix. A pseudograph allows multiple edges between two nodes and self-loops. We prove the correctness of both chains and mixing time for the pseudograph chain by using previous work. The mixing time of the simple graph chain is experimentally evaluated using autocorrelation.

In practice, Monte Carlo Markov Chains are a very popular method for sampling from difficult distributions. However, it is often very difficult to theoretically evaluate the mixing time of the chain, and many practitioners simply stop the chain after 5,000, 10,000 or 20,000 iterations without much justification. Our experimental design with autocorrelation provides a set of statistics that can be used as a justification for choosing a stopping point. Further, we show one way that the autocorrelation technique can be adapted from real-valued samples to combinatorial samples.

6.2 Related Work

The related work can be roughly divided into two categories: constructing and sampling graphs with a fixed degree distribution using sequential importance sampling or Monte Carlo Markov Chain methods, and experimental work on heuristics for generating random graphs with a fixed joint degree distribution.

The methods for constructing graphs with a given degree distribution are primarily either reductions to perfect matchings or sequential sampling methods. There are two popular perfect matching methods. The first is the *configuration model* [30, 7]: k mini-vertices are created for each degree k vertex, and all the mini-vertices are connected. Any perfect matching in the configuration graph corresponds to a graph with the correct degree distribution by merging all of the identified mini-vertices. This allows multiple edges and self-loops, which are often undesirable. See Figure 6.1. The second approach, the *gadget configuration model*, prevents multi-edges and self-loops by creating a gadget for each vertex. If v_i has degree d_i, then it is replaced with a complete bipartite graph (U_i, V_i) with $|U_i| = n - 1 - d_i$ and $|V_i| = n - 1$. Exactly one node in each V_i is connected to each other V_j, representing edge

(i, j) [76]. Any perfect matching in this model corresponds exactly to a simple graph by using the edges in the matching that correspond with edges connecting any V_i to any V_j. We use a natural extension of the first configuration model to the joint degree distribution problem.

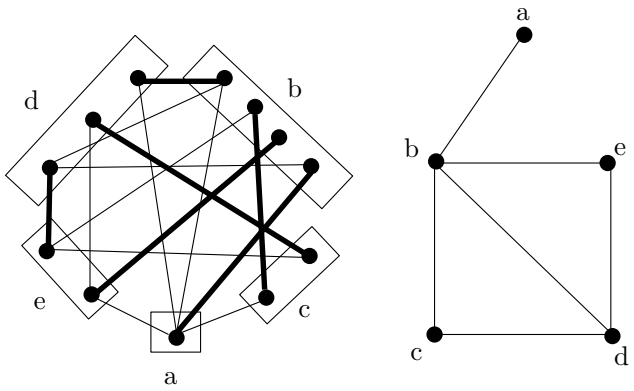

Figure 6.1: On the left, we see an example of the configuration model of the degree distribution of the graph on the right. The edges corresponding to that graph are bold. Each vertex is split into a number of mini-vertices equal to its degree, and then all mini-vertices are connected. Not all edges are shown for clarity.

There are also sequential sampling methods that will construct a graph with a given degree distribution. Some of these are based on the necessary and sufficient Erdős-Gallai conditions for a degree sequence to be graphical [29], while others follow the method of Steger and Wormald [27, 139, 132, 74, 79]. These combine the construction and sampling parts of the problem and can be quite fast. The current best work can sample graphs where $d_{max} = O(m^{1/4-\tau})$ in $O(md_{max})$ time [27].

Another approach for sampling graphs with a given degree distribution is to use a Monte Carlo Markov Chain method. There is significant work on sampling perfect matchings [75, 33]. There has also been work specifically targeted at the degree distribution problem. Kannan, Tetali and Vempala [76] analyze the mixing time of a Markov Chain that mixes on the configuration model, and another for the gadget configuration model. Gkantsidis, Mihail and Zegura [63] use a Markov Chain on the configuration model, but reject any transition that creates a self-loop, multiple edge or disconnects the graph. Both of these chains use the work of Taylor [140] to argue that the state space is connected.

Amanatidis, Green and Mihail study the problems of when a given joint degree matrix has graphical representation and, further, when it has connected graphical representation [12]. They give necessary and sufficient conditions for both of these problems, and constructive algorithms. In Section 2, we give a simpler constructive algorithm for creating a graphical representation that is based on solving the degree sequence problem instead of alternating structures.

Another vein of related work is that of Mahadevan et al. who introduce the concept of dK-series [97, 96]. In this model, d refers to the dimension of the distribution and $2K$ is the joint degree distribution. They propose a heuristic for generating random $2K$-graphs for a fixed $2K$ distribution via edge rewirings. However, their method can get stuck if there exists a degree in the graph for which there is only 1 node with that degree. This is because the state space is not connected. We provide a theoretically sound method of doing this.

Finally, Newman also studies the problem of fixing an assortativity value, finding a *joint remaining degree distribution* with that value, and then sampling a random graph with that distribution using Markov Chains [112, 111]. His Markov Chain starts at any graph with the correct degree distribution and converges to a pseudograph with the correct joint remaining degree distribution. By contrast, our work provides a theoretically sound way of constructing a simple graph with a given joint degree distribution first, and our Markov Chain only has simple graphs with the same joint degree distribution as its state space.

6.3 Notation and Definitions

Formally, a degree distribution of a graph is the probability that a node chosen at random will be of degree k. Similarly, the joint degree distribution is the probability that a randomly selected *edge* will have end points of degree k and l. In this chapter, we are concerned with constructing graphs that exactly match these distributions, so rather than probabilities, we will use a counting definition below and call it the *joint degree matrix*. In particular, we will be concerned with generating *simple* graphs that do not contain multiple edges or self-loops. Any graph that may have multiple edges or self loops will be referred to as a pseudograph.

Definition 1. *The degree vector (DV) $d(G)$ of a graph G is a vector where $d(G)_k$ is the number of nodes of degree k in G.*

A generic degree vector will be denoted by $\mathcal{D}$.

Definition 2. *The joint degree matrix (JDM) $\mathcal{J}(G)$ of a graph G is a matrix where $\mathcal{J}(G)_{k,l}$ is exactly the number of edges between nodes of degree k and degree l in G.*

A generic joint degree matrix will be denoted by $\mathcal{J}$. Given a joint degree matrix, $\mathcal{J}$, we can recover the number of edges in the graph as $m = \sum_{k=1}^{\infty}\sum_{l=k}^{\infty}\mathcal{J}_{k,l}$. We can also recover the degree vector as $\mathcal{D}_k = \frac{1}{k}(\mathcal{J}_{k,k} + \sum_{l=1}^{\infty}\mathcal{J}_{k,l})$. The term $\mathcal{J}_{k,k}$ is added twice because $k\mathcal{D}_k$ is the number of end points of degree k and the edges in $\mathcal{J}_{k,k}$ contribute two end points.

The number of nodes, n is then $\sum_{k=1}^{\infty}\mathcal{D}_k$. This count does not include any degree 0 vertices, as these have no edges in the joint degree matrix. Given n and m, we can easily get the degree distribution and joint degree distribution. They are $P(k) = \frac{1}{n}\mathcal{D}_k$ while $P(k,l) = \frac{1}{m}\mathcal{J}_{k,l}$. Note that $P(k)$ is not quite the marginal of $P(k,l)$ although it is closely related.

The Joint Degree Matrix Configuration Model We propose a new configuration model for the joint degree distribution problem. Given $\mathcal{J}$ and its corresponding $\mathcal{D}$ we create k labeled mini-vertices for every vertex of degree k. In addition, for every edge with end points of degree k and l we create two labeled mini-end points, one of class k and one of class l. We connect all degree k mini-vertices to the class k mini-end points. This forms a complete bipartite graph for each degree, and each of these forms a connected component that is disconnected from all other components. We will call each of these components the "k-neighborhood". Notice that there are $k\mathcal{D}_k$ mini-vertices of degree k, and $k\mathcal{D}_k = \mathcal{J}_{k,k} + \sum_l \mathcal{J}_{k,l}$ corresponding mini-end points in each k-neighborhood. This is pictured in Figure 6.2. Take any perfect matching in this graph. If we merge each pair of mini-end points that correspond to the same edge, we will have some pseudograph that has exactly the desired joint degree matrix. This observation forms the basis of our sampling method.

$$JDM = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 1 & 2 \\ 2 & 2 & 1 \end{pmatrix}$$

$$DV = \begin{pmatrix} 2, & 2, & 2 \end{pmatrix}$$

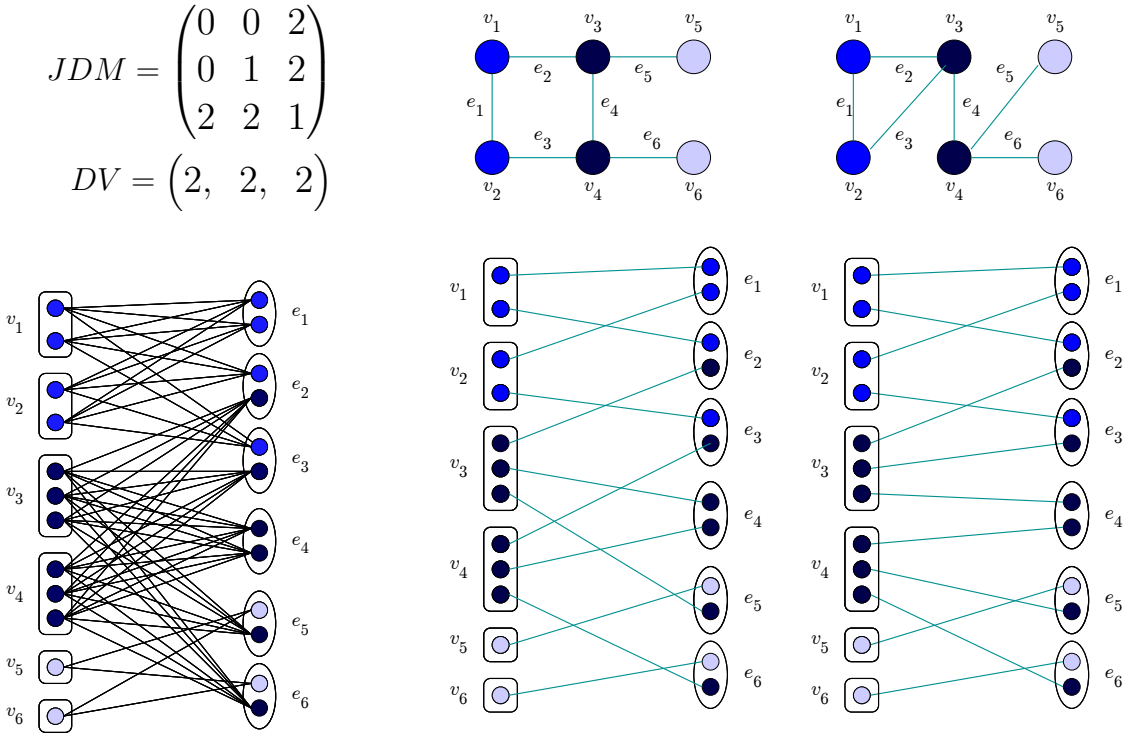

Figure 6.2: The joint degree matrix configuration model. Each vertex is colored according to its degree. On the left is the full model, with the left side consisting of the mini-vertices and the right side of the mini-endpoints. All edges are included, with each of the 3 sets of color vertices forming a complete bipartite graph. The middle and right figures are two realizations of the model, with only the matched edges remaining.

6.4 Constructing Graphs with a Given Joint Degree Matrix

The Erdős-Gallai condition is a necessary and sufficient condition for a degree sequence to be realizable as a simple graph.

Theorem 11. Erdős-Gallai *A degree sequence $\overline{d} = \{d_1, d_2, \cdots d_n\}$ sorted in non-increasing order is graphical if and only if for every $k \leq n$, $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)$.*

The necessity of this condition comes from noting that in a set of vertices of size k, there can be at most $\binom{k}{2}$ internal edges, and for each vertex v not in the subset, there can be at most $\min\{d(v), k\}$ edges entering. The condition considers each subset of decreasing degree vertices and looks at the degree requirements of those nodes. If the requirement is more than the available edges, the sequence cannot be graphical. The sufficiency is shown via the constructive Havel-Hakimi algorithm [68, 67].

The existence of the Erdős-Gallai condition inspires us to ask whether similar necessary and sufficient conditions exist for a joint degree matrix to be graphical. The following necessary and sufficient conditions were independently studied by Amanatidis et al. [12].

Theorem 12. *Let $\mathcal{J}$ be given and $\mathcal{D}$ be the associated degree distribution. $\mathcal{J}$ can be realized as a simple graph if and only if (1) $\mathcal{D}_k$ is integer-valued for all k and (2) $\forall k, l$, if $k \neq l$ then $\mathcal{J}_{k,l} \leq \mathcal{D}_k \mathcal{D}_l$. For each k, $\mathcal{J}_{k,k} \leq \binom{\mathcal{D}_k}{2}$.*

The necessity of these conditions is clear. The first condition requires that there are an integer number of nodes of each degree value. The next two are that the number of edges between nodes of degree k and l (or k and k) are not more than the total possible number of k to l edges in a simple graph defined by the marginal degree sequences. Amanatidis et al. show the sufficiency through a constructive algorithm. We will now introduce a new algorithm that runs in $O(m)$ time.

The algorithm proceeds by building a nearly regular graph for each class of edges, $\mathcal{J}_{k,l}$. Assume that $k \neq l$ for simplicity. Each of the $\mathcal{D}_k$ nodes of degree k receives $\lfloor \mathcal{J}_{k,l}/\mathcal{D}_k \rfloor$ edges, while $\mathcal{J}_{k,l} \mod \mathcal{D}_k$ each have an extra edge. Similarly, the l degree nodes have $\lfloor \mathcal{J}_{k,l}/\mathcal{D}_l \rfloor$ edges, with $\mathcal{J}_{k,l} \mod \mathcal{D}_l$ having 1 extra. We can then construct a simple bipartite graph with this degree sequence. This can be done in linear time in the number of edges using queues as is discussed after Lemma 10. If $k = l$, the only differences are that the graph is no longer bipartite and there are $2\mathcal{J}_{k,k}$ end points to be distributed among $\mathcal{D}_k$ nodes. To find a simple nearly regular graph, one can use the Havel-Hakimi [67, 68] algorithm in $O(\mathcal{J}_{k,k})$ time by using the degree sequence of the graph as input to the algorithm.

We must show that there is a way to combine all of these nearly-regular graphs together without violating any degree constraints. Let $d = \langle d_1, d_2, \cdots d_n \rangle$ be the sorted non-increasing order degree sequence from $\mathcal{D}$. Let $\hat{d}_v$ denote the residual degree sequence where the residual degree of a vertex v is d_v minus the number of edges that currently neighbor v. Also,

let $\hat{\mathcal{D}}_k$ denote the number of nodes of degree k that have non-zero residual degree, i.e. $\hat{\mathcal{D}}_k = \sum_{d_j=k} \mathbf{1}(\hat{d}_j \neq 0)$.

Algorithm 6 Greedy Graph Construction with a Fixed JDM

for $k = n \cdots 1$ **and** $l = k \cdots 1$ **do**
  if $k \neq l$ **then**
    Let $a = \mathcal{J}_{k,l} \mod \mathcal{D}_k$ and $b = \mathcal{J}_{k,l} \mod \mathcal{D}_l$
    Let $x_1 \cdots x_a = \lfloor \frac{\mathcal{J}_{k,l}}{\mathcal{D}_k} \rfloor + 1$, $x_{a+1} \cdots x_{\mathcal{D}_k} = \lfloor \frac{\mathcal{J}_{k,l}}{\mathcal{D}_k} \rfloor$ and $y_1 \cdots y_b = \lfloor \frac{\mathcal{J}_{k,l}}{\mathcal{D}_l} \rfloor + 1$, $y_{b+1} \cdots y_{\mathcal{D}_l} = \lfloor \frac{\mathcal{J}_{k,l}}{\mathcal{D}_l} \rfloor$
    Construct a simple bipartite graph B with degree sequence $x_1 \cdots x_{\mathcal{D}_k}, y_1 \cdots y_{\mathcal{D}_l}$
  else
    Let $c = 2\mathcal{J}_{k,k} \mod \mathcal{D}_k$
    Let $x_1 \cdots x_c = \lfloor \frac{2\mathcal{J}_{k,k}}{\mathcal{D}_k} \rfloor + 1$ and $x_{c+1} \cdots x_{\mathcal{D}_k} = \lfloor \frac{2\mathcal{J}_{k,k}}{\mathcal{D}_k} \rfloor$
    Construct a simple graph B with the degree sequence $x_1 \cdots x_{\mathcal{D}_k}$
  Place B into G by matching the nodes of degree k with higher residual degree with $x_1 \cdots x_a$ and those of degree l with higher residual degree with $y_1 \cdots y_b$. The other vertices in B can be matched in any way with those in G of degree k and l
  Update the residual degrees of each k and l degree node.

To combine the nearly uni

To combine the nearly uniform subgraphs, we start with the largest degree nodes, and the corresponding largest degree classes. It is not necessary to start with the largest, but it simplifies the proof. First, we note that after every iteration, the joint degree sequence is still feasible if $\forall k, l, k \neq l$ $\hat{\mathcal{J}}_{k,l} \leq \hat{\mathcal{D}}_k \hat{\mathcal{D}}_l$ and $\forall k$ $\hat{\mathcal{J}}_{k,k} \leq \binom{\hat{\mathcal{D}}_k}{2}$.

We will prove that Algorithm 6.4 can always satisfy the feasibility conditions. First, we note a fact.

Observation 1. *For all k, $\sum_l \hat{\mathcal{J}}_{k,l} + \hat{\mathcal{J}}_{k,k} = \sum_{d_j=k} \hat{d}_j$*

This follows directly from the fact that the left hand side is summing over all of the k end points needed by $\hat{\mathcal{J}}$ while the right hand side is summing up the available residual end points from the degree distribution. Next, we note that if all residual degrees for degree k nodes are either 0 or 1, then:

Observation 2. *If, for all j such that $d_j = k$, $\hat{d}_j = 0$ or 1 then*
$\sum_{d_j=k} \hat{d}_j = \sum_{d_j=k} \mathbf{1}(\hat{d}_j \neq 0) = \hat{\mathcal{D}}_k$.

Lemma 10. *After every iteration, for every pair of vertices u, v of any degree k,*
$|\hat{d}_u - \hat{d}_v| \leq 1$.

Amanatidis et al. refer to Lemma 10 as the *balanced degree invariant*. This is most easily proven by considering the vertices of degree k as a queue. If there are x edges to be assigned, we can consider the process of deciding how many edges to assign each vertex as being one

of popping vertices from the top of the queue and reinserting them at the end x times. Each vertex is assigned edges equal to the number of times it was popped. The next time we assign edges with end points of degree k, we start with the queue at the same position as where we ended previously. It is clear that no vertex can be popped twice without all other vertices being popped at least once.

Lemma 11. *The above algorithm can always greedily produce a graph that satisfies $\mathcal{J}$, provided $\mathcal{J}$ satisfies the initial necessary conditions.*

Proof. There is one key observation about this algorithm - it maximizes $\hat{\mathcal{D}}_k\hat{\mathcal{D}}_l$ by ensuring that the residual degrees of any two vertices of the same degree never differ by more than 1. By maximizing the number of available vertices, we can not get stuck adding a self-loop or multiple edge. From this, we gather that if, for some degree k, there exists a vertex j such that $\hat{d}_j = 0$, then for all vertices of degree k, their residuals must be either 0 or 1. This means that $\sum_{d_j=k} \hat{d}_j = \hat{\mathcal{D}}_k \geq \hat{\mathcal{J}}_{k,l}$ for every other l from Observation 2.

From the initial conditions, we have that for every k, l $\mathcal{J}_{k,l} \leq \mathcal{D}_k\mathcal{D}_l$. $\mathcal{D}_k = \hat{\mathcal{D}}_k$ provided that all degree k vertices have non-zero residuals. Otherwise, for any unprocessed pair, $\mathcal{J}_{k,l} \leq \min\{\hat{\mathcal{D}}_k, \hat{\mathcal{D}}_l\} \leq \hat{\mathcal{D}}_k\hat{\mathcal{D}}_l$. For the k, k case, it is clear that $\mathcal{J}_{k,k} \leq \hat{\mathcal{D}}_k \leq \binom{\hat{\mathcal{D}}_k}{2}$. Therefore, the residual joint degree matrix and degree sequence will always be feasible, and the algorithm can always continue. □ □

A natural question is that since the joint degree distribution contains all of the information in the degree distribution, do the joint degree distribution necessary conditions easily imply the Erdős-Gallai condition? This can easily be shown to be true.

Corollary 1. *The necessary conditions for a joint degree matrix to be graphical imply that the associated degree vector satisfies the Erdős-Gallai condition.*

6.5 Uniformly Sampling Graphs with Monte Carlo Markov Chain (MCMC) Methods

We now turn our attention to uniformly sampling graphs with a given graphical joint degree matrix using MCMC methods. We return to the joint degree matrix configuration model. We can obtain a starting configuration for any graphical joint degree matrix by using Algorithm 1. This configuration consists of one complete bipartite component for each degree with a perfect matching selected. The transitions we use select any end point uniformly at random, then select any other end point in its degree neighborhood and swap the two edges that these neighbor. In Figure 6.2, this is equivalent to selecting one of the square endpoints uniformly at random and then selecting another uniformly at random from the same connected component and then swapping the edges. A more complex version of this chain checks that this swap does not create a multiple edge or self-loop. Formally, the transition function is a randomized algorithm given by Algorithm 7.

Algorithm 7 Markov Chain Transition Function

1: With probability 0.5, stay at configuration C. Else:
2: Select any endpoint e_1 uniformly at random. It neighbors a vertex v_1 in configuration C
3: Select any e_2 u.a.r from e_1's degree neighborhood. It neighbors v_2
4: (Optional: If the graph obtained from the configuration with edges $E \cup \{(e_1, v_2), (e_2, v_1)\} \setminus \{(e_1, v_1), (e_2, v_2)\}$ contains a multi-edge or self-loop, reject)
5: $E \leftarrow E \cup \{(e_1, v_2), (e_2, v_1)\} \setminus \{(e_1, v_1), (e_2, v_2)\}$

There are two chains described by Algorithm 7. The first, $\mathcal{A}$ doesn't have the optional step and its state space is all pseudographs with the desired joint degree matrix. The second, $\mathcal{B}$ includes the optional step and only transitions to and from simple graphs with the correct joint degree matrix.

We remind the reader of the standard result that any irreducible, aperiodic Markov Chain with symmetric transitions converges to the uniform distribution over its state space. For details, see Chapter 7 of [107]. Both $\mathcal{A}$ and $\mathcal{B}$ are aperiodic, due to the self-loop to each state. From the description of the transition function, we can see that $\mathcal{A}$ is symmetric. This is less clear for the transition function of $\mathcal{B}$. Is it possible for two connected configurations to have a different number of feasible transitions in a given degree neighborhood? We show that it is not the case in the following lemma.

Lemma 12. *The transition function of $\mathcal{B}$ is symmetric.*

Proof. Let C_1 and C_2 be two neighboring configurations in $\mathcal{B}$. This means that they differ by exactly 4 edges in exactly 1 degree neighborhood. Let this degree be k and let these edges be e_1v_1 and e_2v_2 in C_1 whereas they are e_1v_2 and e_2v_1 in C_2. We want to show that C_1 and C_2 have exactly the same number of feasible k-degree swaps.

Without loss of generality, let e_x, e_y be a swap that is prevented by e_1 in C_1 but allowed in C_2. This must mean that e_x neighbors v_1 and e_y neighbors some $v_y \neq v_1, v_2$. Notice that the swap e_1e_x is currently feasible. However, in C_2, it is now infeasible to swap e_1, e_x, even though e_x and e_y are now possible.

If we consider the other cases, like e_x, e_y is prevented by both e_1 and e_2, then after swapping e_1 and e_2, e_x, e_y is still infeasible. If swapping e_1 and e_2 makes something feasible in C_1 infeasible in C_2, then we can use the above argument in reverse. This means that the number of feasible swaps in a k-neighborhood is invariant under k-degree swaps. □ □

The remaining important question is the connectivity of the state space over these chains. It is simple to show that the state space of $\mathcal{A}$ is connected. We note that it is a standard result that all perfect matchings in a complete bipartite graph are connected via edge swaps [140]. Moreover, the space of pseudographs can be seen exactly as the set of all perfect matchings over the disconnected complete bipartite degree neighborhoods in the joint degree matrix configuration model. The connectivity result is much less obvious for $\mathcal{B}$. In particular, the difficulty lies in the fact that transitions like those in Figure 6.4 can not be made without

going through a state that results in a pseudograph. We adapt a result of Taylor [140] that all graphs with a given degree sequence are connected via edge swaps in order to prove this. The proof is inductive and follows the structure of Taylor's proof.

Theorem 13. *Given two simple graphs, G_1 and G_2 of the same size with the same joint degree matrix, there exists a series of end point rewirings to transform G_1 into G_2 (and vice versa) where every intermediate graph is also simple.*

Proof. This proof will proceed by induction on the number of nodes in the graph. The base case is when there are 3 nodes. There are 3 realizable JDMs. Each is uniquely realizable, so there are no switchings available.

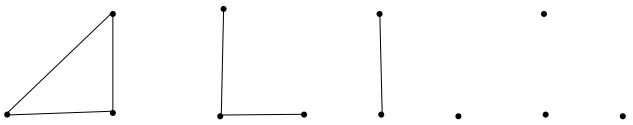

Figure 6.3: The four potential joint degree distributions when $n = 3$.

Assume that this is true for $n = k$. Let G_1 and G_2 have $k + 1$ vertices. Label the nodes of G_1 and G_2 $v_1 \cdots v_{k+1}$ such that $deg(v_1) \geq deg(v_2) \geq \cdots \geq deg(v_{k+1})$. Our goal will be to show that both graphs can be transformed in G_1' and G_2' respectively such that v_1 neighbors the same nodes in each graph, and the transitions are all through simple graphs. Now we can remove v_1 to create G_1'' and G_2'', each with $n - 1$ nodes and identical JDMs. By the inductive hypothesis, these can be transformed into one other and the result follows.

We will break the analysis into two cases. For both cases, we will have a set of target edges, $e_1, e_2 \cdots e_{d_1}$ that we want v_1 to be connected to. Without loss of generality, we let this set be the edges that v_1 currently neighbors in G_2. We assume that the edges are ordered in reverse lexicographic order by the degrees of their end points. This will guarantee that the resulting construction for v_1 is graphical and that we have a non-increasing ordering on the requisite end points. Now, let k_i denote the end point in G_2 for edge e_i that isn't v_1.

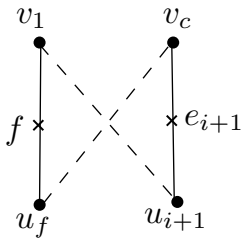

Figure 6.4: The dotted edges represent the troublesome edges that we may need to swap out before we can swap v_1 and v_c.

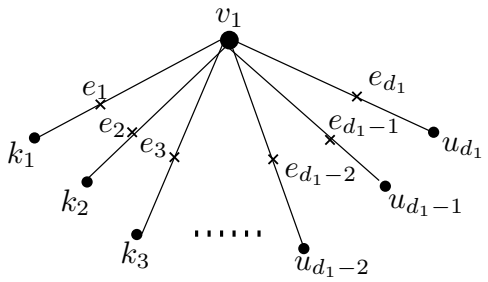

Figure 6.5: The disk is v_1. The crosses are the end points correctly neighbored, $e_1 \cdots e_{d_1}$.

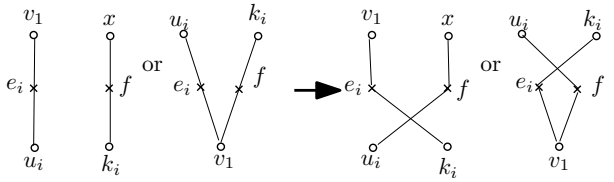

Figure 6.6: The two parts of Case (1).

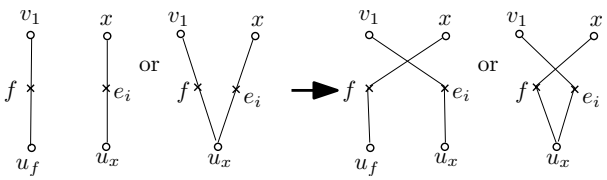

Figure 6.7: The two parts of Case (2)

Case 1) For the first case, we will assume that v_1 is already the end point of all edges $e_1, e_2 \cdots e_{d_1}$ but that all of the k_i may not be assigned correctly as in Figure 6.5. Assume that $e_1, e_2 \cdots e_{i-1}$ are all edges $(v_1, k_1) \cdots (v_1, k_{i-1})$ and that e_i is the first that isn't matched to its appropriate k_i.

Call the current end point of the other end point of e_i u_i. We know that $deg(k_i) = deg(u_i)$ and that k_i currently neighbors $deg(k_i)$ other nodes, $\Gamma(k_i)$. We have two cases here. One is that $v_1 \in \Gamma(k_i)$ but via edge f instead of e_i. Here, we can swap v_1 on the end points of f and e_i so that the edge $v_1 - e_i - k_i$ is in the graph. f can not be an e_j where $j < i$ because those edges have their correct end points, k_j assigned. This is demonstrated in Figure 6.6.

The other case is that $v_1 \notin \Gamma(k_i)$. If this is the case, then there must exist some $x \in \Gamma(k_i) \setminus \Gamma(u_i)$ because $d(u_i) = d(k_i)$ and u_i neighbors v_1 while k_i doesn't. Therefore, we can swap the edges $v_1 - e_i - u_i$ and $x - f - k_i$ to $v_1 - e_i - k_i$ and $x - f - u_i$ without creating any self-loops or multiple edges. This is demonstrated in Figure 6.6.

Therefore, we can swap all of the correct end points onto the correct edges.

Case 2) For the second case, we assume that the edges $e_1, \cdots e_{d_1}$ are distributed over l nodes of degree d_1. We want to show that we can move all of the edges $e_1 \cdots e_{d_1}$ so that v_1 is an end point. If this is achievable, we have exactly Case 1.

Let $e_1, \cdots e_{i-1}$ be currently matched to v_i and let e_i be matched to some x such that $deg(x) = d_1$. Let f be an edge currently matched to v_1 that is not part of $e_1 \cdots e_{d_1}$ and let its other end point be u_f. Let the other end point of e_i be u_x as in Figure 6.7.

We now have several initial cases that are all easy to handle. First, if v, x, u_x, u_f are all distinct and (v, u_x) and (x, u_f) are not edges then we can easily swap v and x such that the edges go from $v - f - u_f$ and $x - e_i - u_x$ to $v - e_i - u_x$ and $x - f - u_f$. Next, if $u_f = u_x$ then we can simply swap v_1 onto e_i and x onto f and, again, v_1 will neighbor e_i. This will not create any self-loops or multiple edges because the graph itself will be isomorphic. This situations are both shown in Figure 6.7.

The next case is that $x = u_f$. If we try to swap v_1 onto e_i then we create a self-loop from x to x via f. Instead, we note that since the JDM is graphical, there must exist a third vertex y of the same degree as v_1 and x that does not neighbor x. Now, y neighbors an edge g, and we can swap $x - f$ and $y - g$ to $x - g$ and $y - f$. The edges are $v_1 - f - y$ and $x - e_i - u_i$ and e_i can be swapped onto v_1 without conflict.

The cases left to analyze are those where the nodes are all distinct and (v_1, u_x) or (x, u_f) are edges in the graph. We will analyze these separately.

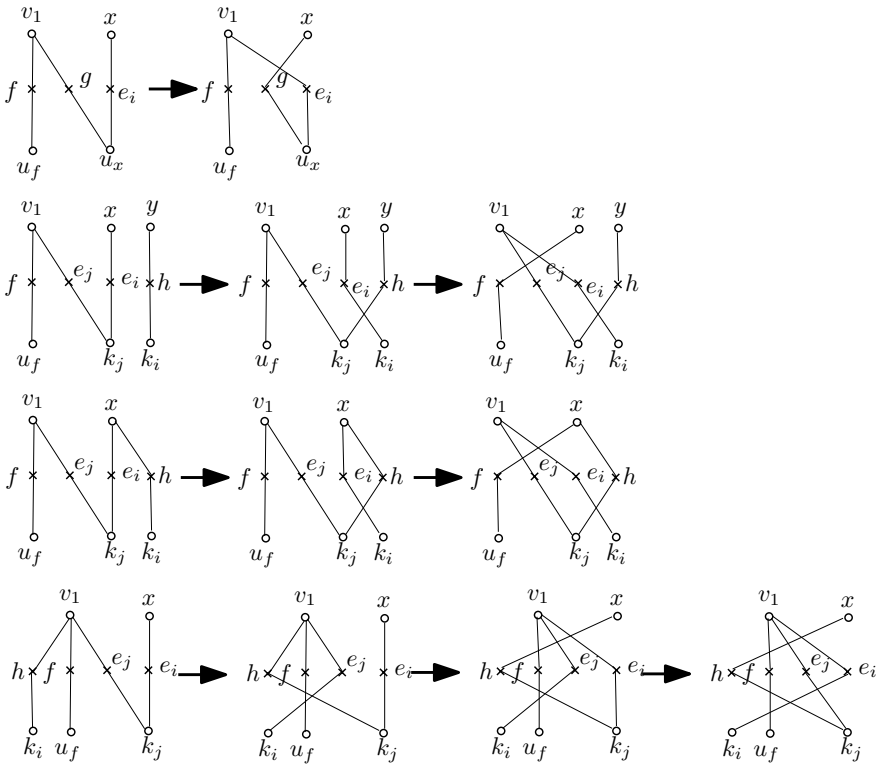

Figure 6.8: A graphical representation of the situations discussed in Case (2a).

Case 2a) If (v_1, u_x) is an edge in the graph, then it must be so through some edge named g. Note that this means we have $v_1 - g - u_x$ and $x - e_i - u_x$. We can swap this to $v_1 - e_i - u_x$ and $x - g - u_x$ and have an isomorphic graph provided that g is not some e_j where $j < i$. This is the top case in Figure 6.8.

If g is some e_j then it must be that $u_x = k_j$. This is distinct from k_i. $deg(k_j) = deg(k_i)$ so there must exist some edge h that k_i neighbors with its other end point being y. There are again three cases, when $y \neq x, v_1$ $y = x$ and when $y = v_1$. These are the bottom three rows illustrated in Figure 6.8. The first is the simplest. Here, we can assume that k_j does not neighbor y (because it neighbors v_1 and x that k_i does not) so we can swap k_j onto h and k_i onto e_1. This has removed the offending edge, and we can now swap v_1 onto e_1 and x onto f.

When $y = x$, we first swap k_i onto e_j and k_j onto h. Next, we swap v onto e_i and x onto f as they no longer share an offending edge.

Finally, when $y = v_1$, we use a sequence of three swaps. The first is k_i onto e_j and k_j onto h. The next is v_1 onto e_1 and x onto h. Finally, we swap k_j back onto e_j and k_i onto e_i.

Case 2b) If (x, u_f) is an edge in the graph, then it must be through some edge g such that $x - g - u_f$ and $x - e_i - u_x$. Without loss of generality, assume that f is the only edge

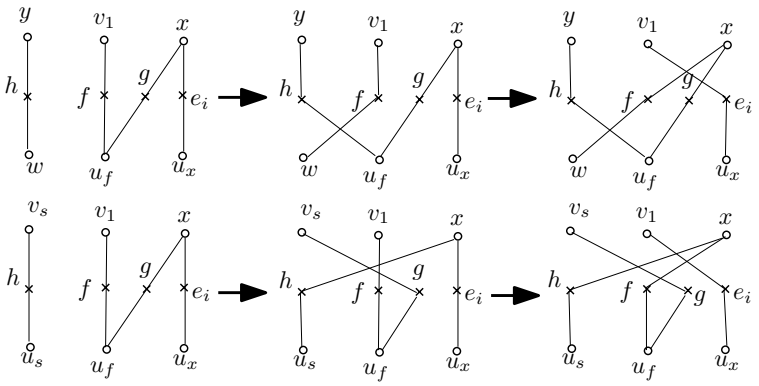

Figure 6.9: A graphical representation of the situations discussed in Case (2b)

neighboring v_1 that isn't an e_j. Since f doesn't neighbor v_1 in G_2, there must either exist a w with $deg(w) = deg(u_f)$ or v_s with $deg(v_s) = d(v_1)$. This relies critically upon the fact that f and g are the same class edge. If there is a w, then it doesn't neighbor v_1 (or we can apply the above argument to find a w') and it must have some neighbor $y \in \Gamma(w) \setminus \Gamma(u)$ through edge h. Therefore, we can swap u_f onto h and w onto f. This removes the offending edge, and we can now swap v_1 onto e_i and x onto f.

If v_s exists instead, then by the same argument, there exists some edge h with end point u_s such that $v_s \notin \Gamma(u_f)$ and $u_s \notin \Gamma(x)$. Therefore, we can swap $v_s - h$ and $x - g$ to $v_s - g$ and $x - h$. This again removes the troublesome edge and allows us to swap v_1 onto e_i.

Therefore, given any node, a precise set of edges that it should neighbor, and a set of vertices that are the end points of those edges, we can use half-edge-rewirings to transform any graph G to G' that has this property, provided the set of edges is graphical. □ □

Now that we have shown that both $\mathcal{A}$ and $\mathcal{B}$ converge to the uniform distribution over their respective state spaces, the next question is how quickly this happens. Note that from the proof that the state space of $\mathcal{B}$ is connected, we can upper bound the diameter of the state space by $3m$. The diameter provides a lower bound on the mixing time. In the next section, we will empirically estimate the mixing time to be also linear in m.

6.6 Estimating the Mixing Time of the Markov Chain

The Markov chain $\mathcal{A}$ is very similar to one analyzed by Kannan, Tetali and Vempala [76]. We can exactly use their canonical paths and analysis to show that the mixing time is polynomial. This result follows directly from Theorem 3.2 and Corollary 3.2 (or Theorem 3 and Corollary 4 or Theorem 4.2 and Corollary 4.2) of [76] for chain $\mathcal{A}$. This is because the joint degree matrix configuration model can be viewed as $|\mathcal{D}|$ complete, bipartite, and disjoint components. These components should remain disjoint, so the Markov Chain can be viewed as a 'meta-chain' which samples a component and then runs one step of the Kannan,

Tetali and Vempala chain on that component. Even though the mixing time for this chain is provably polynomial, this upper bound is too large to be useful in practice.

The analysis to bound the mixing time for chain $\mathcal{B}$ is significantly more complicated. One approach is to use the canonical path method to bound the congestion of this chain. The standard trick is to define a path from G_1 to G_2 that fixes the misplaced edges identified by $G_1 \oplus G_2$, the symmetric difference between the two graphs, in a globally ordered way. However, this is difficult to apply to chain $\mathcal{B}$ because fixing a specific edge may not be atomic, i.e. from the proof of Theorem 13 it may take up to 4 swaps to correctly connect a vertex with an end point if there are conflicts with the other degree neighborhoods. These swaps take place in other degree neighborhoods and are not local moves. Therefore, this introduces new errors that must be fixed, but can not be incorporated into $G_1 \oplus G_2$. In addition, step (4) also prevents us from using path coupling as a proof of the mixing time.

Given that bounding the mixing time of this chain seems to be difficult without new techniques or ideas, we use a series of experiments that substitute the *autocorrelation time* for the mixing time.

6.6.1 Autocorrelation Time

Autocorrelation time is a quantity that is related to the mixing time and is popular among physicists. We will give a brief introduction to this concept, and refer the reader to Sokal's lecture notes for further details and discussion [134].

The autocorrelation of a signal is the cross-correlation of the signal with itself given a lag t. More formally, given a series of data $\langle X_i \rangle$ where each X_i is a drawn from the same distribution X with mean μ and variance σ, the autocorrelation function is $R_X(t) = \frac{E[(X_i - \mu)(X_{i-t} - \mu)]}{\sigma^2}$.

Intuitively, the inherent problem with using a Markov Chain sampling method is that successive states generated by the chain may be highly correlated. If we were able to draw independent samples from the stationary distribution, then the autocorrelation of that set of samples with itself would go to 0 as the number of samples increased. The autocorrelation time is capturing the size of the gaps between sampled states of the chain needed before the autocorrelation of this 'thinned' chain is very small. If the thinned chain has 0 autocorrelation, then it must be exactly sampled from the stationary distribution. In practice, when estimating the autocorrelation from a finite number of samples, we do not expect it to go to exactly 0, but we do expect it to 'die away' as the number of samples and gap increases.

Definition 3. *The exponential autocorrelation time is* $\tau_{exp,X} = \limsup_{t \to \infty} \frac{t}{-\log |R_X(t)|}$ *[134].*

Definition 4. *The integrated autocorrelation time is* $\tau_{int,X} = \frac{1}{2} \sum_{t=-\infty}^{\infty} R_X(t) = \frac{1}{2} + \sum_{t=1}^{\infty} R_X(t)$ *[134].*

The difference between the exponential autocorrelation time and the integrated autocorrelation time is that the exponential autocorrelation time measures the time it takes for the chain to reach equilibrium after a cold start, or 'burn-in' time. The integrated autocorrelation time is related to the increase in the variance over the samples from the Markov Chain

as opposed to samples that are truly independent. Often, these measurements are the same, although this is not necessarily true.

We can substitute the autocorrelation time for the mixing time because they are measuring very similar things - the number of iterations that the Markov Chain needs to run for before the difference between the current distribution and the stationary distribution is small. However, it is impossible to actually measure the mixing time. We will use the integrated autocorrelation time estimate.

6.6.2 Experimental Design

We used the Markov Chain $\mathcal{B}$ in two different ways. First, for each of the smaller datasets, we ran the chain for 50,000 iterations 15 times. We used this to calculate the autocorrelation values for each edge for each lag between 100 and 15,000 in multiples of 100. From this, we calculated the estimated integrated autocorrelation time, as well as the iteration time for the autocorrelation of each edge to drop under a threshold of 0.001. This is discussed in Section 6.6.4.

We also replicated the experimental design of Raftery and Lewis [120]. Given our estimates of the autocorrelation time for each size graph in Section 6.6.4, we ran the chain again for long enough to capture 10,000 samples where each sample had x iterations of the chain between them. x was chosen to vary from much smaller than the estimated autocorrelation time, to much larger. From these samples, we calculated the sample mean for each edge, and compared it with the actual mean from the joint degree matrix. We looked at the total variational distance between the sample means and actual means and showed that the difference appears to be converging to 0. We chose the mean as an evaluation metric because we were able to calculate the true means theoretically. We are unaware of another similarly simple metric.

We used the formulas for empirical evaluation of mixing time from page 14 of Sokal's survey [134]. In particular, we used the following:

- The sample mean is $\overline{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$.
- The sample unnormalized autocorrelation function is $\hat{C}(t) = \frac{1}{n-t}\sum_{i=1}^{n-t}(x_i - \overline{\mu})(x_{i+t} - \overline{\mu})$.
- The natural estimator of $R_X(t)$ is $\hat{\rho}(t) = \hat{C}(t)/\hat{C}(0)$
- The estimator for $\tau_{int,X}$ is $\hat{\tau}_{int} = \frac{1}{2}\sum_{t=-(n-1)}^{n-1}\lambda(t)\hat{\rho}(t)$ where λ is a 'suitable' cutoff function.

Data Sets We have used several publicly available datasets, Word Adjacencies [113], Les Miserables [81], American College Football [62], the Karate Club [151], the Dolphin Social Network [95], C. Elegans Neural Network (celegans) [148, 149], Power grid (power) [148], Astrophysics collaborations (astro-ph) [110], High-Energy Theory collaborations (hep-th) [110], Coauthorships in network science (netscience) [113], and a snapshot of the Internet from 2006

(as-22july) [114]. In the following $|V|$ is the number of nodes, $|E|$ is the number of edges and $|\mathcal{J}|$ is the number of non-zero entries in the joint degree matrix.

Dataset	$\|E\|$	$\|V\|$	$\|\mathcal{J}\|$
AdjNoun	425	112	159
as-22july	48,436	22,962	5,496
astro-ph	121,251	16,705	11,360
celegans	2,359	296	642
Dolphins	159	62	61
Football	616	115	18
hep-th	15,751	8,360	629
Karate	78	34	40
LesMis	254	77	99
netscience	2,742	1,588	184
power	6,594	4,940	108

Table 6.1: Details about the datasets, $|V|$ is the number of nodes, $|E|$ is the number of edges and $|\mathcal{J}|$ is the number of unique entries in the $\mathcal{J}$.

6.6.3 Relationship Between Mean of an Edge and Autocorrelation

For each of the smaller graphs, AdjNoun, Dolphins, Football, Karate and LesMis, we ran the Markov Chain 10 times for 50,000 iterations and collected an indicator variable for each potential edge. For each of these edges, and each run, we calculated the autocorrelation function for values of t between 100 and 15,000 in multiples of 100. For each edge, and each run, we looked at the t value where the autocorrelation function first dropped below the threshold of 0.001. We then plotted the mean of these values against the mean of the edge, i.e. if it connects vertices of degree d_i and d_j (where $d_i \neq d_j$) then $\mu_e = \mathcal{J}_{d_i,d_j}/d_i d_j$ or $\mu_e = \mathcal{J}_{d_i,d_i}/\binom{d_i}{2}$ otherwise. The three most useful plots are given in Figures 6.10 and 6.11 as the other graphs did not contain a large range of mean values.

From these results, we identified a potential relationship between μ_e and the time to pass under a threshold. Unfortunately, none of our datasets contained a significant number of edges with larger μ_e values, i.e. between 0.5 and 1. In order to test this hypothesis, we designed a synthetic dataset that contained the many edges with values of μ_e at $\frac{i}{20}$ for $i = 1, \cdots 20$.

Designing Synthetic Data Our goal was to represent all of the potential means for $\frac{i}{20}$ for $0 < i \leq 20$. We note that 20 factors into 4 and 5, so we want to first fix some degrees

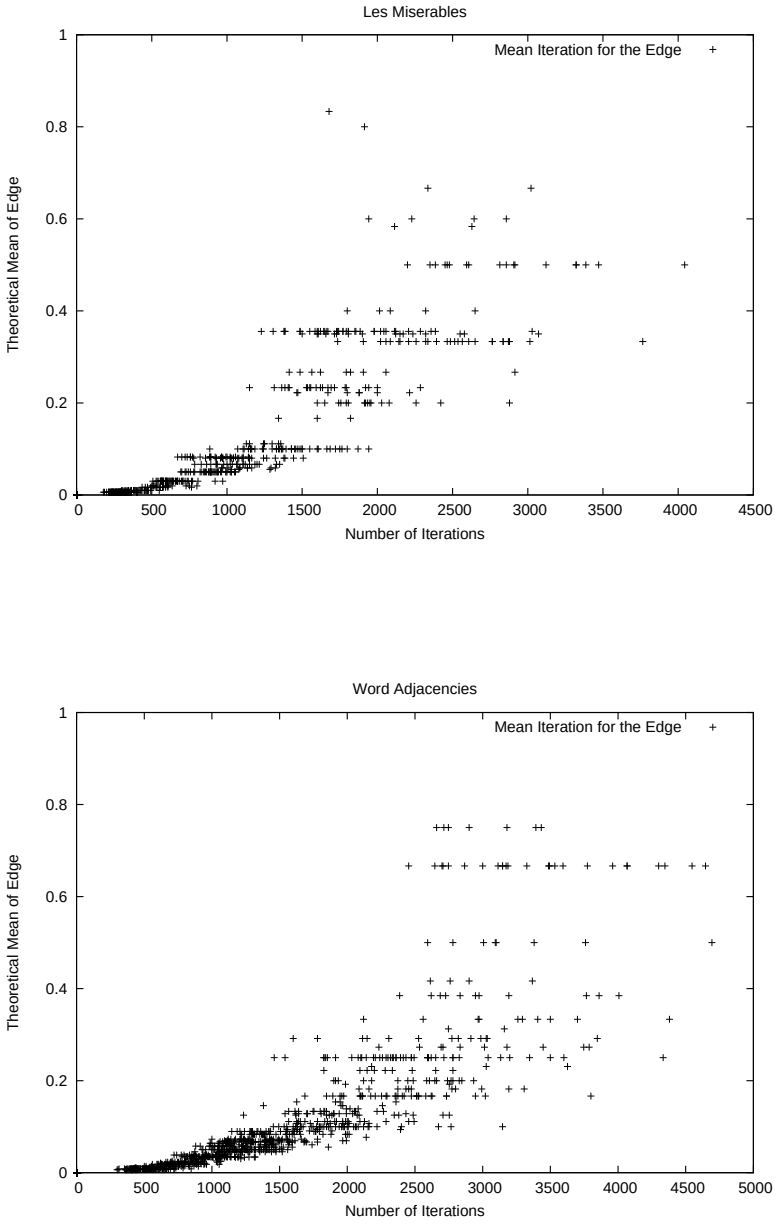

Figure 6.10: The time for an edge's estimated autocorrelation function to pass under the threshold of 0.001 versus μ_e for that edge for LesMis and AdjNoun from top to bottom.

such that $\mathcal{D}_k = 4$ and $\mathcal{D}_l = 5$. For convenience, because the maximum number of edges we will be assigning is 20, we will pick these degrees to be $K = \{20, 21, 22, 23, 24\}$ for $\mathcal{D}_k = 4$ and $L = \{25, 26, 27, 28\}$ for $\mathcal{D}_l = 5$. The number of each we picked was to guarantee that there were at least 20 combinations of edge types. We can now assign the values $1 - 20$ arbitrarily to $\mathcal{J}_{K\times L}$. This assignment clearly satisfies that $\mathcal{J}_{k,l} \leq \mathcal{D}_k\mathcal{D}_l$ so far.

Now, we must fill in the rest of $\mathcal{J}$ so that $\mathcal{D}$ is integer valued for degrees. One way is to note that we should have 4×20 degree 20 edges. We can sum the number of currently allocated edges with one end point of degree 20, call this x and set $\mathcal{J}_{1,20} = 80 - x$. There are many other ways of consistently completing $\mathcal{J}$, such as assigning as many edges as possible to the $K \times K$ and $L \times L$ entries, like $\mathcal{J}_{20,21}$. This results in a denser graph. For the synthetic graph used in this chapter, we completed $\mathcal{J}$ by adding all edges as $(1, 20), (1, 21)$ etc edges. We chose this because it was simple to verify and it also made it easy to ignore the edges that were not of interest.

The final dataset we created had 326 edges, 194 vertices and 21 distinct $\mathcal{J}$ entries. We ran the Markov Chain 200 times for this synthetic graph. For each run, we calculated the threshold value for each edge. Figure 6.11 shows the edges' mean vs its mean time for the autocorrelation value to pass under 0.001. We see that there is a roughly symmetric curve that obtains its maximum at $\mu_e = 0.5$.

This result suggests a way to estimate the autocorrelation time for larger graphs without repeating the entire experiment for every edge that could possibly appear. One can calculate μ_e for each edge from the JDM and sample edges with μ_e around 0.5. We use this method for selecting our subset of edges to analyze. In particular, we sampled about 300 edges from each of the larger graphs. For all of these except for power, the μ_e values were between 0.4 and 0.6. For power, the maximum μ_e value is about 0.15, so we selected edges with the largest μ values.

6.6.4 Autocorrelation Values

For each dataset and each run we calculated the unnormalized autocorrelation values. For the smaller graphs, this entailed setting t to every value between 100 and 15,000 in multiples of 100. We randomly selected 1 run for each dataset and graphed the autocorrelation values for each of the edges. We present the data for the Karate and Dolphins datasets in Figures 6.12 and 6.13. For the larger graphs, we changed the starting and ending points, based on the graph size. For example, for Netscience was analyzed from 2,000 to 15,000 in multiples of 100, while as-22july was analyzed from 1,000 to 500,000 in multiples of 1,000.

All of the graphs exhibit the same behavior. We see an exponential drop off initially, and then the autocorrelation values oscillate around 0. This behavior is due to the limited number of samples, and a bias due to using the sample mean for each edge. If we ignore the noisy tail, then we estimate that the autocorrelation 'dies off' at the point where the mean absolute value of the autocorrelation approximately converges, then we can locate the 'elbow' in the graphs. This estimate for all graphs is given in Table 6.6.7 at the end of this Section.

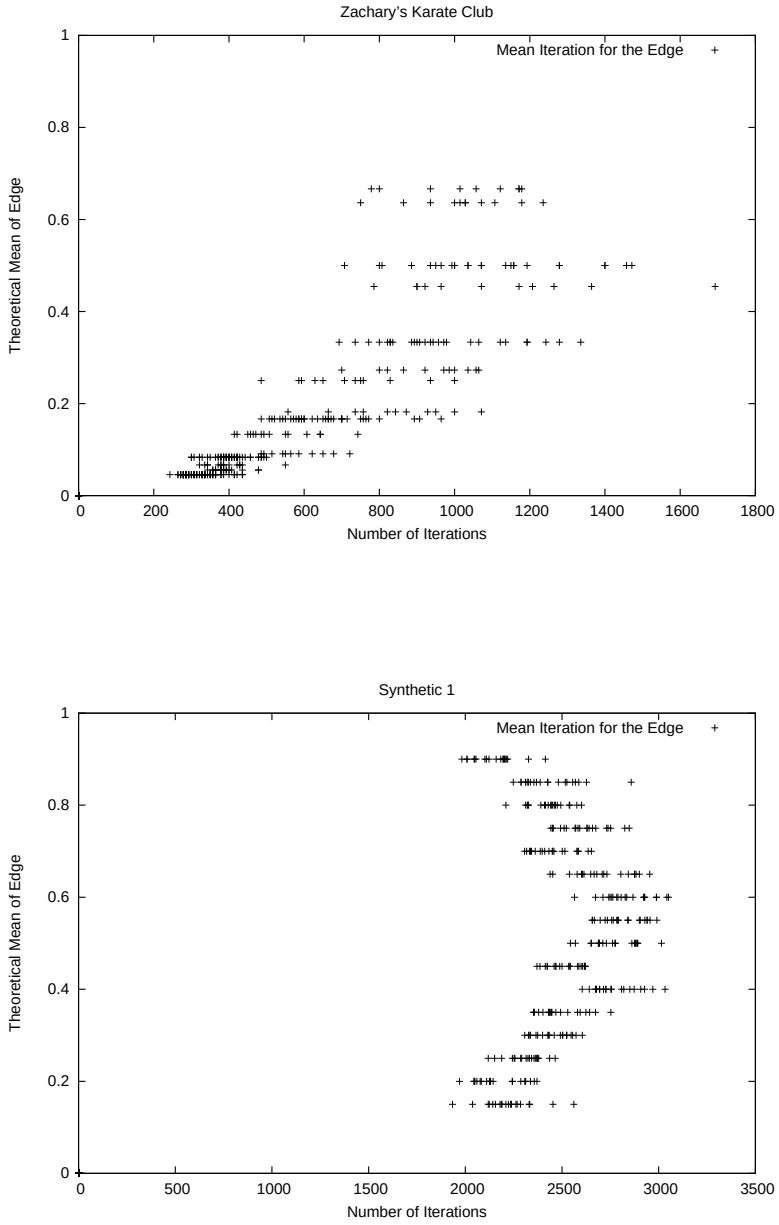

Figure 6.11: The time for an edge's estimated autocorrelation function to pass under the threshold of 0.001 versus μ_e for that edge for Karate and the synthetic dataset. The synthetic dataset has a larger range of μ_e values than the real datasets and a significant number of edges for each value.

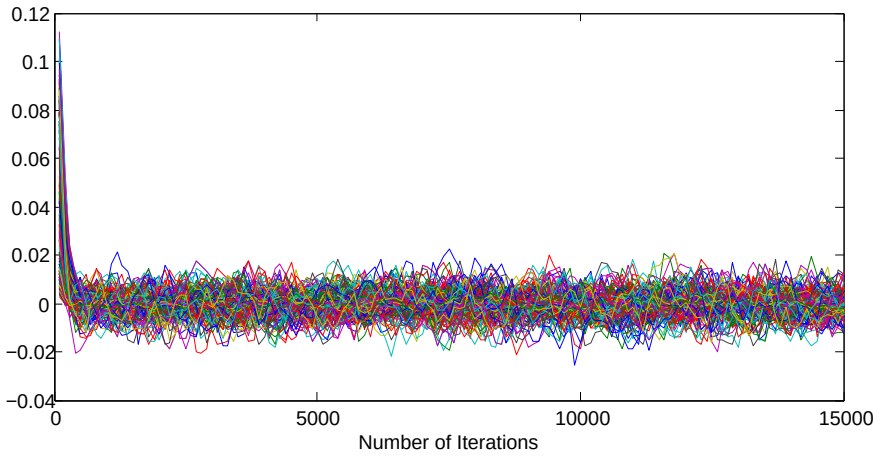

Figure 6.12: The exponential drop-off for Karate appears to end after 400 iterations.

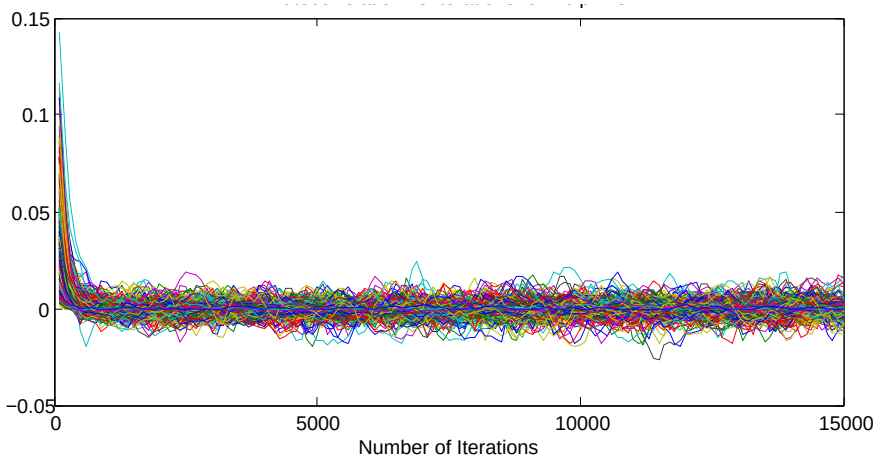

Figure 6.13: The exponential drop-off for Dolphins appears to end after 600 iterations.

6.6.5 Estimated Integrated Autocorrelation Time

For each dataset and run, we calculated the estimated integrated autocorrelation time. For the datasets with fewer than 1,000 edges, we calculated the autocorrelation in lags of 100 from 100 to 15,000 for each dataset. For the larger ones, we used intervals that depended on the total size of the graph. We estimate $\hat{\rho}(t)$ as the size of the intervals times the sum of the values. The cut-off function we used for the smaller graphs was $\lambda(t) = 1$ if $0 < t < 15,000$ and 0 otherwise. This value was calculated for each edge. In Table 6.6.5 we present the mean, maximum and minimum estimated integrated autocorrelation time for each dataset over the runs of the Markov Chain using three different methods. For each of the edges, we first calculated the mean, median and max estimated integrated autocorrelation value over the various runs. Then, for each of these three values for each edge, we calculated the max, mean and min over all edges. For each of the graphs, the data series representing the median and max have each had their x-values perturbed slightly for clarity.

These values are graphed on a log-log scale plot. Further, we also present a graph showing the ratio of these values to the number of edges. The ratio plot, Figure 6.15, suggests that the autocorrelation time may be a linear function of the number of edges in the graph, however the estimates are noisy due to the limited number of runs.

All three metrics give roughly the same picture. We note that there is much higher variance in estimated autocorrelation time for the larger graphs. If we consider the evidence of the log-log plot and the ratio plot, we suspect that the autocorrelation time of this Markov Chain is linear in the number of edges.

6.6.6 The Sample Mean Approaches the Real Mean for Each Edge

Given the results of the previous experiment estimating the integrated autocorrelation time, we next executed an experiment suggested by Raftery and Lewis [120]. First we note that for each edge e, we know the true value of $P(e \in G | G \text{ has } \mathcal{J})$ is exactly $\frac{\mathcal{J}_{k,l}}{\mathcal{D}_k \mathcal{D}_l}$ or $\frac{\mathcal{J}_{k,k}}{\binom{\mathcal{D}_k}{2}}$ if e is an edge between degrees k and l. This is because there are $\mathcal{D}_k \mathcal{D}_l$ potential (k, l) edges that show up in any graph with a fixed $\mathcal{J}$, and each graph has $\mathcal{J}_{k,l}$ of them. If we consider the graphs as being labeled, then we can see that each edge has an equal probability of showing up when we consider permutations of the orderings.

Thus, our experiment was to take samples at varying intervals, and consider how the sample mean of each edge compared with our known theoretical mean. For the smaller graphs, we took 10,000 samples at varying gaps depending on our estimated integrated autocorrelation time and repeated this 10 times. Additionally, we saw that the total variational distance quickly converged to a small, but non-zero value. We repeated this experiment with 20,000 samples and, for the two smallest graphs, Karate and Dolphins, we repeated the experiment with 5,000 and 40,000 samples. These results show that this error is due to the number of samples and not the sampler. For the graphs with more than 1,000 edges, each run resulted

A summary of the Estimate Integrated Autocorrelation Times

Dataset	$\|E\|$	mean: mean	max	min
Karate	78	288.92	444.1	221.13
Dolphins	159	383.21	553.84	256.13
LesMis	254	559.77	931.35	129.45
AdjNoun	425	688.71	1154.9	156.49
Football	616	962.42	2016.9	404.77
celegans	2,359	3340.2	4851.4	2458.8
netscience	2,742	1791.4	3147.2	1087.7
power	6,594	6624.5	17933	2166.9
hep-th	15,751	26552	36816	14976
as-22july	48,436	89637	139280	60627
astro-ph	121,251	121860	298970	37706
		median: mean	max	min
Karate	78	288.31	443	217.63
Dolphins	159	377.4	550.99	211.44
LesMis	254	542.43	895.57	57.492
AdjNoun	425	659.06	1160.3	66.851
Football	616	925.97	1646.9	349.12
celegans	2,359	3235.7	4861.4	2323.6
netscience	2,742	1658.3	3033.2	937.8382
power	6,594	4768.8	16901	250.6012
hep-th	15,751	25608	37004	14130
as-22july	48,436	87190	152490	58493
astro-ph	121,251	119900	321730	46830
		maximum: mean	max	min
Karate	78	382.59	608.06	268.95
Dolphins	159	528.86	1134.1	397.35
LesMis	254	894.08	2598.6	342.76
AdjNoun	425	1186.1	4083.6	350.97
Football	616	1546.4	7514.3	967
celegans	2,359	4844.6	7836.9	3065.5
netscience	2,742	3401	7404	1894.7
power	6,594	20599	54814	7074.7
hep-th	15,751	46309	64936	25753
as-22july	48,436	121930	256520	76214
astro-ph	121,251	152930	408000	84498

Table 6.2: Mean refers to taking the mean autocorrelation time for each edge, and then the mean, min and max of these values over all measured edges. Similarly, the next set of results is the median for each edge, with the min, mean and max reported. Finally, maximum is the max for each edge, again with the mean, min and max reported.

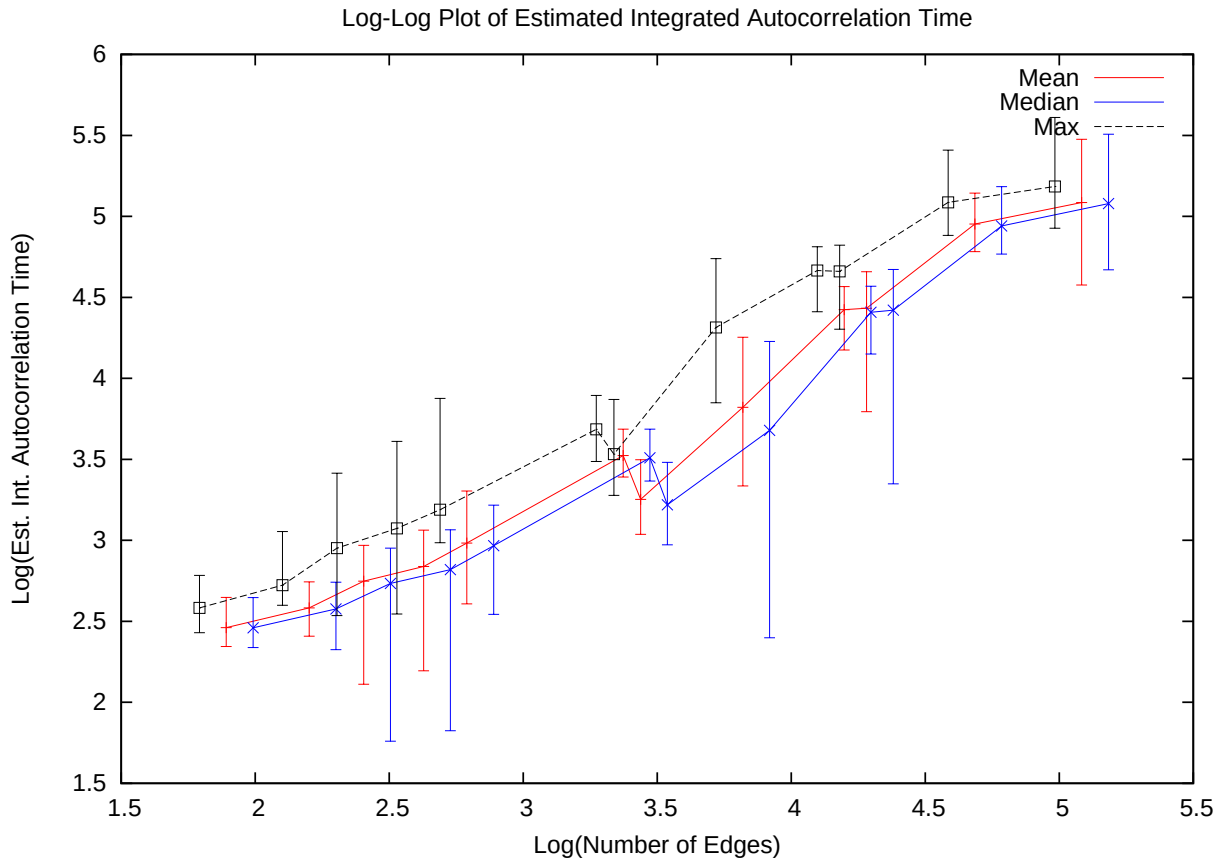

Figure 6.14: The max, median and min values over the edges for the estimated integrated autocorrelation times in a log-log plot. L to R in order of size: Karate, Dolphins, LesMis, AdjNoun, Football, celegans, netscience, power, hep-th, as-22july and astro-ph

in 20,000 samples at varying gaps, and this was repeated 5 times. We present these results in Figures 18 through 28. If $S_{e,g}$ is the sample mean for edge e and gap g, and μ_e is the true mean, then the graphed value is $\sum_e |S_{e,g} - \mu_e| / \sum_e \mu_e$.

In all of the figures, the line runs through the median error for the runs and the error bars are the maximum and minimum values. We note that the maximum and minimum are very close to the median as they are within 0.05% for most intervals. These graphs imply that we are sampling uniformly after a gap of 175 for the Karate graph. For the dolphin graph, we see very similar results, and note that the error becomes constant after a sampling gap of 400 iterations.

For the larger graphs, we varied the gaps based on the graph size, and then focused on the area where the error appeared to be decreasing. Again, we see consistent results, although the residual error is higher. This is to be expected because there are more potential edges in these graphs, so we took relatively fewer samples per edge. A summary of the results can be found in Table 6.6.7.

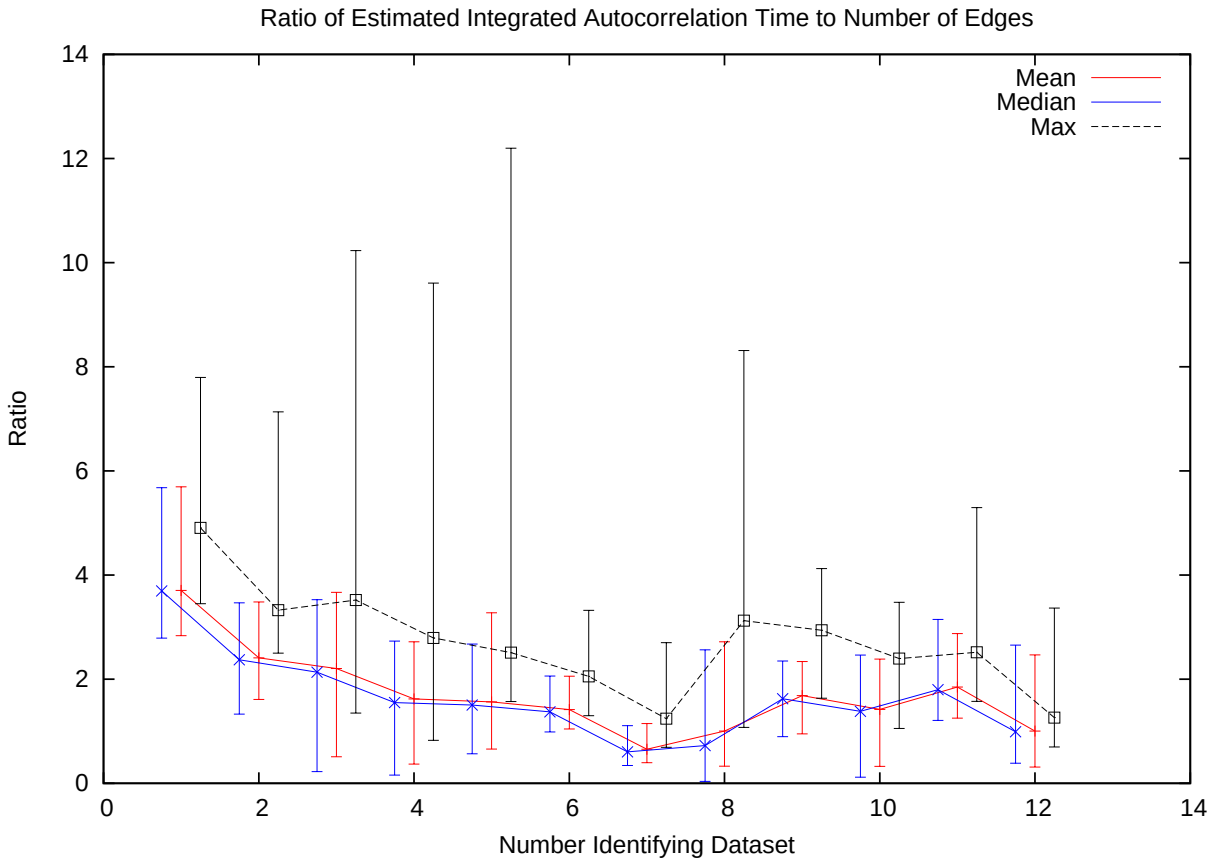

Figure 6.15: The ratio of the max, median and min values over the edges to the number of edges for the estimated integrated autocorrelation times. L to R in order of size: Karate, Dolphins, LesMis, AdjNoun, Football, celegans, netscience, power, hep-th, as-22july and astro-ph

6.6.7 Summary of Experiments

Based on the results in this table, our recommendation would be that running the Markov Chain for $5m$ steps would satisfy all running time estimates except for Power's results for the Maximum Estimated Integrated Autocorrelation time. This estimate is significantly lower than the result for Chain $\mathcal{A}$ that was obtained using the standard theoretical technique of canonical paths.

6.7 Conclusions

This chapter makes two primary contributions. The first is the investigation of Markov Chain methods for uniformly sampling graphs with a fixed joint degree distribution. Previous work shows that the mixing time of $\mathcal{A}$ is polynomial, while our experiments suggest that the

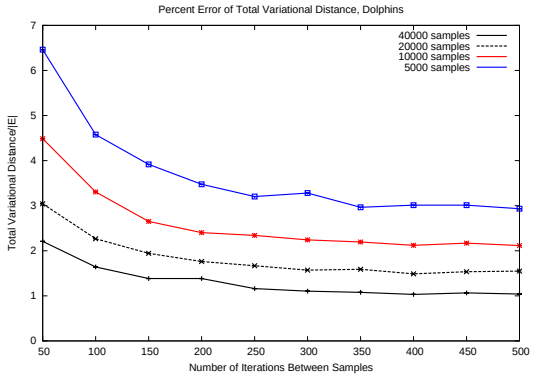

Figure 6.16: The Dolphin Dataset with 5,000 to 40,000 samples

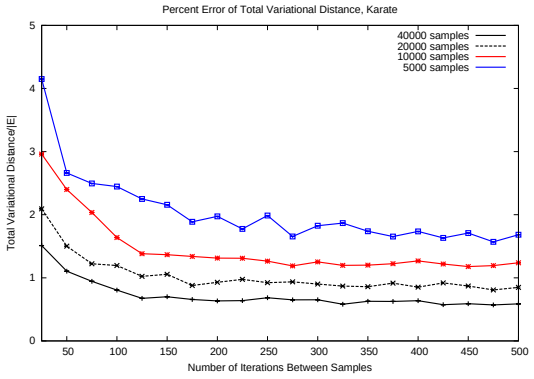

Figure 6.17: The Karate Dataset with 5,000 to 40,000 samples

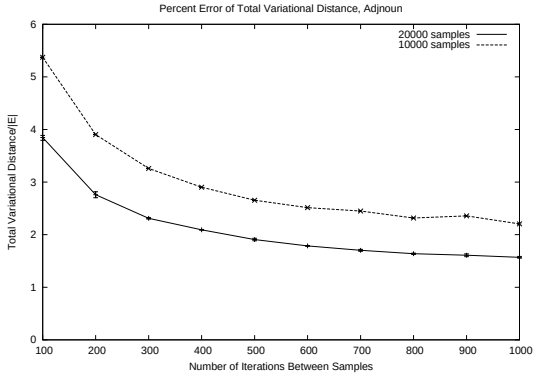

Figure 6.18: The AdjNoun Dataset with 10,000 and 20,000 samples

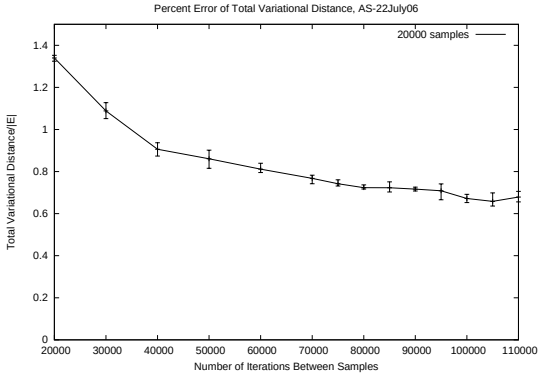

Figure 6.19: The AS-22July06 Dataset with 20,000 samples

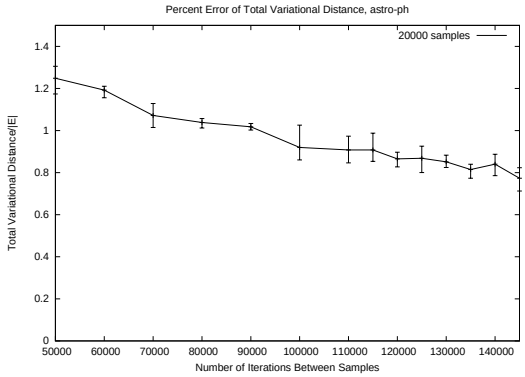

Figure 6.20: The Astro-PH Dataset with 20,000 samples

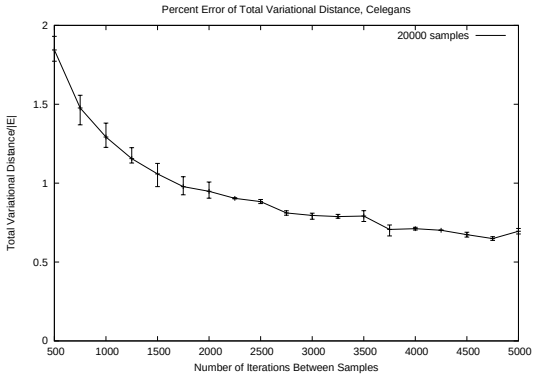

Figure 6.21: The Celegans Dataset with 20,000 samples

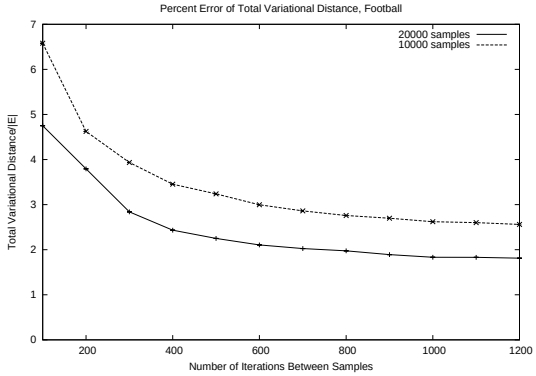

Figure 6.22: The Football Dataset with 10,000 and 20,000 samples

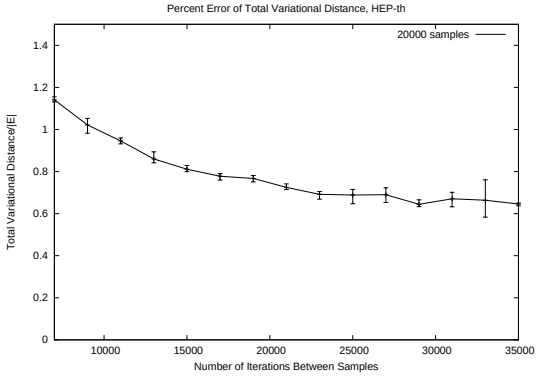

Figure 6.23: The Hep-TH Dataset with 20,000 samples

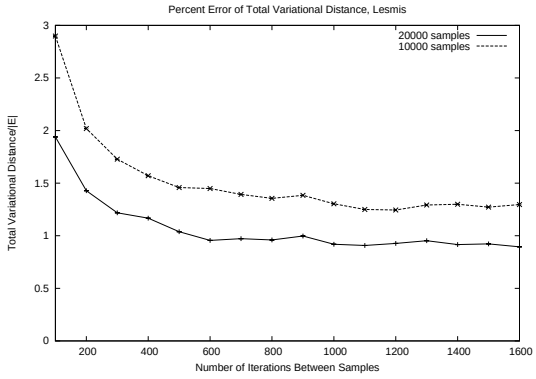

Figure 6.24: The LesMis Dataset with 10,000 and 20,000 samples

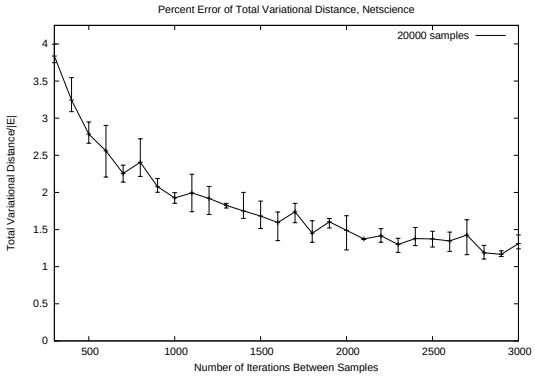

Figure 6.25: The Netscience Dataset with 20,000 samples

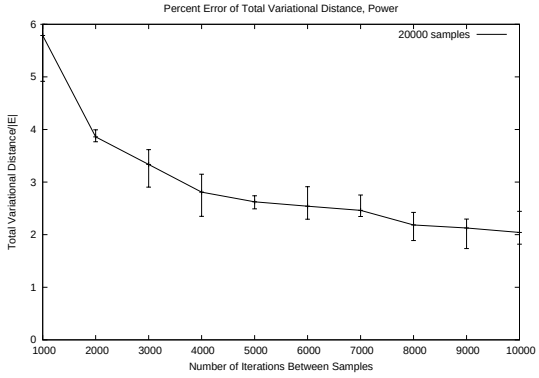

Figure 6.26: The Power Dataset with 20,000 samples

Summary of Estimates

	$\|E\|$	Max EI	Mean Conv.	Thresh.
AdjNoun	425	1,186	900	700
AS-22July	48,436	256,520	95,000	156,744
Astro-PH	121,251	408,000	120,000	343,154
Celegans	2,359	7,836.9	3,750	7,691
Dolphins	159	528	400	600
Football	616	1,546	1,000	900
Hep-TH	15,751	64,936	28,000	22,397
Karate	78	382	175	400
LesMis	254	894	800	1,000
Netscience	2,742	7,404	2,000	7,017
Power	6,594	54,814	8,000	7,270

Table 6.3: The values are the Maximum Estimated Integrated Autocorrelation time (Max EI, the third column of Table 6.6.5), the Sample Mean Convergence iteration number, and the time to drop under the Autocorrelation Threshold. The Autocorrelation threshold was calculated as when the average absolute value of the autocorrelation was less than 0.0001

mixing time of $\mathcal{B}$ is also polynomial. The relationship between the mean of an edge and the autocorrelation values can be used to efficiently experiment with larger graphs by sampling edges with mean between 0.4 and 0.6 and repeating the analysis for just those edges. This was used to repeat the experiments for larger graphs and to provide further convincing evidence of polynomial mixing time.

Our second contribution is in the design of the experiments to evaluate the mixing time of the Markov Chain. In practice, it seems the stopping time for sampling is often chosen without justification. Autocorrelation is a simple metric to use, and can be strong evidence that a chain is close to the stationary distribution when used correctly.

Chapter 7

An Introduction to Tournaments

As a natural way to select a leader, competition is at the heart of life. It is intriguing, both for its participants, and its spectators. Society is riddled with organized competitions called *tournaments* with well-defined rules to select a winner from a pool of candidate players. Sports tournaments such as the FIFA World Cup and Wimbledon are immensely popular and generate huge amounts of revenue. Elections are another important type of tournaments: a leading party is selected according to some rules using votes from the population.

Sports tournaments and voting mechanisms share the same primary goal - to identify a socially optimal or 'best' candidate. They also share many of the same constraints, both in terms of being designed so that incentives are aligned correctly, and in terms of execution time and complexity. The primary difference between tournaments and elections is that in a tournament all of the candidates exactly coincide with the voters, and the outcome of each match is determined only by the players involved, not by all of the voters. Because of this, the question of whether the tournament is designed in a way that aligns the incentives of players with the goal of the mechanism is a bit easier to quantify - is it ever in a players' interest to lie about the outcome of a match, i.e. intentionally lose? When this happens in professional sporting events, as it did in the 2012 Olympics Badminton tournament, spectators become extremely upset even if the players' goal is just to eventually win.

Another feature that makes tournaments quite different from the standard voting mechanism setting is noise. It would be very unusual to assume that the votes themselves are noisy representations of the ordering of a voters' opinion of the candidates. However, we observe noisy outcomes in sports matches all the time from stories of the underdog football team winning against the state champions one time in twenty years, to the design of the Major League Baseball World Series where the first team to win the majority of seven games is declared the victor. Thus, tournaments must fundamentally deal with minimizing the effect of noisy outcomes in addition to identifying a strong player with the minimum number of games.

7.1 The Tournament Graph

Using slightly overloaded terminology, the tournament graph represents the outcomes of all possible matches between players. More specifically, it is a complete directed graph over all n players in the tournament. The edge directed from u to v exists if u would beat v in a match, and (v, u) exists otherwise. The edges of the tournament graph can be weighted, usually with the probability of u winning against v if they were to face each other.

The tournament graph has several interesting features. The first is the existence of kings. A player v is a kind in a tournament graph $G = (V, E)$ if it is the case that for all other players $u \in V$, it is either the case that v beats u, i.e. $(v, u) \in E$, or v beats a player y who beats u, i.e. $(v, y), (y, u) \in E$. Every tournament graph has at least one king as the player with the highest outdegree.

Outside of the study of tournament competitions, tournament graphs have been widely analyzed. It is much more difficult to prove NP-hardness for these types of graphs for many problems because of the volume of edges involved. Conversely, it is often the case that easy approximation algorithms exist specifically for tournament graphs. For example, feedback arc set has a 3 approximation in tournament graphs by using the ordering resulting from the outdegree of the nodes [44].

7.2 Types of Tournaments

In this section, we will formally introduce several popular tournament designs and discuss their advantages and drawbacks.

Round-Robin A round-robin tournament, also known as an 'all-play-all' tournament, is one where every player meets every other player. Each team may play each other exactly once or, rarely, exactly twice. For the single match setting, the outcome of the tournament can exactly be represented by the tournament graph. If there is exactly one player with the most number of wins, they are declared the victor. If not, a tie-breaking method must be used.

There are several advantages to the round-robin format. The first is that it addresses the problem of noisy outcomes by allowing each player to play $n-1$ (or $2(n-1)$) total matches so one bad performance need not eliminate a strong player. Additionally, it allows every player to continuing participating for the entirety of the tournament. This is not done to improve the selection aspect of the mechanism but instead to encourage people to participate when considerable expense and time has been invested in attending the tournament.

By the same token, the fact that every player must face every other player can also be viewed as a disadvantage. This format require $\binom{n}{2}$ games and enough space for $\frac{n}{2}$ matches to occur simultaneously for $n-1$ rounds. Space can be traded in exchange for taking more time. Additionally, for any player who is not competitive for the top spot, playing all of the extra games is unnecessary in order to determine this. Round-robin is often used in tournaments

as a qualifying round with the top k finishers moving on to the finals. Unfortunately, when a particular team knows they have already placed within the top k, the format no longer encourages them to play hard and win their final matches. This can occur even when not used in a qualifying round - if a match occurring late in the tournament occurs between a team that still has a chance of winning and one who does not, then the second team has no incentive to try as well.

Swiss-system Tournaments The Swiss-system tournament design addresses the problem of the time required to play a round-robin tournament. Again, all players play for the entirety of the tournament, but the tournament consists of pre-determined number of rounds. After each round, each player is awarded 1 point for winning a match (and possibly 0.5 for tying). The matches for the next round are determined by matching players against others with the same score (or closest score if that is not possible) subject to constraints like no two players should play twice. After the pre-determined number of rounds, if there is a player with a unique highest score, they are declared the winner. If not, some tie-breaking procedure must be used. There are many different systems used to determine the matches at each round.

The Swiss-system has the same advantage as round-robin of guaranteeing that all players play all rounds while requiring many fewer games (approximately $n \log n$) to find the result. Additionally, it avoids the problems of pitting strong players against weak players in the final rounds of the tournament where the matches are both uninteresting to the spectators and demoralizing for the weaker team. It is also able to generate an approximate ranking of the strength of all of the players based on their final score. This ranking is generally most meaningful for the top and bottom players, but not those in the middle. This helps to motivate all players to play well, even when it is clear they can not become the final winner. Unfortunately, this can also be counted as a disadvantage - the strongest player may have built up a 2-point lead entering the final round, guaranteeing them a win despite the outcome of this round. This is significantly less exciting for spectators when the strongest player is not motivated to try their best.

Knock-out Tournaments The primary focus of the next two chapters, a knock-out tournament allows some number of players to be eliminated after each round. The most common of these tournaments is known as the *single-elimination tournament* where a player is eliminated after a single loss. Also used is a *double-elimination tournament* where each player is allowed two losses before being eliminated.

There are two major aspects of these types of tournaments, the bracket which is the ordering for which players meet each other when depending on the previous outcomes, and the existence of *byes*. A bye grants a player an automatic win in a round and is needed when there is an uneven number of players in the round. Often, byes will be granted to an appropriate number of players in the first round to make the number of players in the second round a power of 2. Several of the rounds have distinctive names - when 8 players remain, the round is the Quarter-Finals, 4 players participate in the Semi-Finals and 2 players reach

the Final.

Knock-out Tournaments take significantly fewer rounds and matches than round-robin tournaments for a total of $\log n$ rounds for single-elimination and $2 \log n$ for double-elimination. By contrast, the Swiss system usually requires at least $\log n$ rounds as well. As the number of players shrinks dramatically after each round (half of all players are eliminated after the first round in single-elimination) it can solve space constraints nicely. Unfortunately, the main disadvantage is that it is very susceptible to noisy outcomes. If a very strong player has a bad game in the first round, this can totally remove them from the competition. Additionally, single-elimination can be a very frustrating format for amateur competitors as they do not participate for very long. The double-elimination tournament addresses this slightly by allowing ever player to participate in at least two games.

There are many ways to seed the initial bracket for knock-out tournaments. The most popular when a previous ranking is known is to seed the strongest player against the weakest, the second strongest against the second weakest etc. Round-robin tournaments are often used during the season to compete for these seeding positions, leading to strange behavior as teams jockey over seeding positions for the playoffs.

7.2.1 Tournaments as represented in the tournament graph

As previously mentioned, the outcomes of the round-robin tournament as exactly represented by a tournament graph. Similarly, if given a tournament graph representing the match outcomes and a seeding of a single-elimination tournament, the single-elimination tournament can be represented as a *spanning binomial arborescence* in the tournament graph, rooted at the winner of the tournament.

A spanning binomial arborescence is defined recursively. A binomial arboresence of size 2 is exactly 1 directed edge between two vertices. The root of the tree is the source of the edge. A binomial arboresence of size n consists of 2 binomial arboresences of size $\frac{n}{2}$ with a directed edge between the roots of the two sub-arboresences. This connection is the key feature of the work in the following chapters.

7.3 The Gibbard-Satterthwaite Theorem and Tournaments

One of the most celebrated results in social choice theory is Arrow's Impossibility Theorem [19] which can be summarized as "there is no good voting mechanism". More precisely, it assumes axioms that any good voting mechanism must satisfy:

- Non-dictatorship: The social welfare function should not be the function of a single voter's preferences.
- Universality: For any set of voter preferences, the social choice function should yield a single complete ranking of the candidates deterministically.

- Independence of Irrelevant Alternatives: The social choice function's ranking of x and y should depend solely on the individual voter's rankings of x and y.
- Monotonicity: If any individual changes their ranking to place x above y, then the social choice function should not change the outcome to now place y above x.
- Non-imposition: Every possible ranking should be achievable as the result of some set of input preferences.

Arrow's Theorem shows that no voting mechanism can satisfy all axioms. While this result can be taken in a negative light, it also has allowed us to develop a rich theory containing a diversity of voting mechanisms, each appropriate for a different setting.

An extension of Arrow's theorem, discovered by Allan Gibbard [61] and Mark Satterthwaite [128], states that for three or more candidates, it is necessarily true that any voting mechanism is either:

- The mechanism is dictatorial
- There is some candidate who can never win
- The rule is susceptible to tactical voting or *manipulation*

By manipulation, the authors mean a situation where one voter can submit a false preference profile and the voting mechanism will now generate an outcome that is more preferable to that user. As a concrete example, consider 3 people, Alice, Bob and Charlie, who are voting on how to spend an afternoon - watching a movie, going sailing, or going to an art museum. Alice's vote is sailing > movie > art museum, Bob prefers movie > art museum > sailing and Charlie prefers art museum > sailing > movie. If we use a Borda count scheme where 2 points is awarded for each first preference and 1 for the second, each alternative receives a total of 3 points. If the mechanisms tie break scheme is to prefer Alice's vote, then the outcome is sailing > movie > art museum. If, on the other hand, Bob submits a false vote of art museum > movie > sailing, then the final scores are sailing - 3 points, movie - 2 points and art museum - 4 points and a final ranking of art museum > sailing > movie. This is a better outcome for both Bob and Charlie but worse for Alice.

While the above example is somewhat contrived because of the tie-breaking scheme used, the result of the theorem is that such an example can be constructed for every 'reasonable' voting mechanism. In fact, this is exactly the dilemma faced by American voters when choosing to vote for a third party candidate - it may be the case that by not voting for their second choice candidate, their third choice wins, as notably happened in the Florida vote totals for the 2000 Presidential Election. Note that the Gibbard-Satterthwaite Theorem does not claim that all outcomes can be manipulated, just that there exists situations where the preferences are such that they can be.

In 1989, Bartholdi, Tovey and Trick [24] introduced the idea of quantifying the difficulty of manipulation using the framework of computational complexity. While it is always the

case that manipulation can exist, if it is the case that even when it does it is computational intractable for an adversary to compute a manipulation, then maybe a mechanism isn't really susceptible. This observation began the entire subarea of computational social choice. To date, we have a multitude of results of the form "manipulating mechanism x is NP-hard" as well as algorithms for finding manipulations when they exist in polynomial time.

Manipulation in Tournaments The Gibbard-Satterthwaite Theorem, as stated, only applies to a single voter changing their vote. In the context of tournaments, this corresponds to a player choosing to lose a game they otherwise would win. It seems that this would mean that tournaments are safe from manipulation, provided the only preferences are that each player prefers they win and doesn't care who else does, and also that the tournament is designed so that no player can do better by losing a match. Unfortunately, this is not true as tournaments allow many other kinds of manipulation.

The first type of manipulation is by a coalition of players. While it may be true that no single player would prefer to lose a match, for a coalition, the goal is to guarantee that one of their members wins. To this end, they may be able to strategically select a set of matches to lose in order to propel their preferred candidate forward.

Another type of manipulation is *agenda control*. Here, the manipulation is performed by the tournament organizer. The organizer has the power to select the seeding of the players and the ordering of the matches. This does not affect the outcome in a round-robin tournament, but it certainly can for a Swiss-style or knockout tournament where a players path is heavily influenced by an early win or loss.

A third type of manipulation is *destructive manipulation*. Previously, we have only mentioned cases where the goal is to make a candidate a winner. It is also possible to manipulate in order to prevent a certain player from becoming the winner.

Chapter 8

Rigging a Tournament

Two of the most common tournament formats employed in both sports and voting are *round-robin* and *single-elimination*. In the former, every pair of players are matched up, and a player's score is how many matches they won. If some player has beaten everyone else, then they are the clear (Condorcet) winner. Otherwise, the winner is not well-defined. However, given the outcomes of a round-robin tournament, there are various methods of producing rankings of the players. The most common definition of the optimal ranking is that it minimizes the number of wins of a lower-ranked player over a higher-ranked player [133]. Although finding such a ranking for a round-robin tournament is NP-hard [11], sorting the players according to their number of wins is a good approximation to the optimum ranking [44].

Single-elimination (SE) tournaments are played as follows. First, a permutation of the players, called the *bracket* or *schedule* is given. According to the bracket, the first two players are matched up, then the second pair of players etc. The winners of the matches move on to the next round. The bracket for this round is obtained by pairing up the remaining players according to the original bracket. If the number of players is a power of 2, the tournament is balanced. Otherwise, it is unbalanced and some players advance to the next round without playing a match. In practice, these *byes* are usually granted in the first round. Although the winner of an SE tournament is always well-defined, the chance of a particular player winning the tournament can vary immensely depending on the bracket. Arguably, this gives the tournament organizer a lot of power. The study of how much control an organizer has over the outcome of a tournament is called *agenda control* [25].

The most studied agenda control problem for balanced SE tournaments is to find a bracket which maximizes the probability that a given player will win the tournament. The tournament organizer is given the probability that i will beat j for every pair of players i, j. A major focus is to maximize the winning probability of the *strongest* player under some assumptions[1] (e.g., [17, 72, 146, 145]). Without assumptions on the probabilities, the agenda control problem for an arbitrary given player is NP-hard [86, 69], even when the probabilities

[1] A common assumption is monotonicity: the probability of beating a weaker player is at least as high as that of beating a stronger one.

are in $\{0, 1, 1/2\}$ [144]. Moreover, the maximum probability that a given player wins cannot be approximated within any constant factor unless P=NP [144]. When the probabilities are all either 0 or 1, the agenda control problem, then called the tournament fixing problem (TFP), is not well understood. One of the interesting open problems in computational social choice is whether a tournament fixing bracket can be efficiently found. Several variants of the problem are NP-hard – when some pairs of players cannot be matched [143], when some players must appear in given rounds [144], or when the most "interesting" tournament is to be computed [86].

Besides its natural connection to tournament manipulation, TFP studies the relationship between round-robin and single-elimination tournaments. The decision version of TFP asks, given the results of a round-robin tournament and a player $\mathcal{A}$, is $\mathcal{A}$ also the winner of some SE tournament, given the same match outcomes? In the area of voting, suppose all votes are in, can we simulate a win for a particular candidate, using single-elimination rules (binary cup)? In this work, we investigate the following question: if we consider a round-robin tournament and a ranking produced from it by sorting the players according to their number of wins, how many of the top players can actually win some SE tournament, given the same match outcomes? What conditions on the round-robin tournament suffice so that one can efficiently rig the SE tournament outcome for many of the top players?

Prior work has shown several intuitive results. For instance, if $\mathcal{A}$ is any player with the maximum number of wins in a round-robin tournament, then one can efficiently construct a winning (balanced) SE bracket for $\mathcal{A}$ [143]. We extend and strengthen many of the prior results.

Contributions. Let Π be an ordering of the players in nonincreasing order of their number of wins in the given round-robin tournament. We consider conditions under which, for large K, the SE tournament can be fixed efficiently for *any* of the first K players in Π. We are interested in natural and not too restrictive conditions under which a constant fraction of the players can be made to win. If the first player p_1 in Π beats everyone else, then p_1 wins all SE tournaments. We show that if *any* player can beat p_1, then we can also fix the tournament for the second player p_2. We show that for large enough tournaments, if there is a matching onto the top $K-1$ players $\{p_1, \ldots, p_{K-1}\}$ in Π from the rest of the players, then we can efficiently find a bracket for which p_K wins, where K is as large as 19% of the players.

Graph representation. The outcome of a round-robin tournament has a natural graph representation as a *tournament graph*: a directed graph in which for every pair of nodes a, b, there is an edge either from a to b, or from b to a. The nodes of a tournament graph represent the players in a round-robin tournament, and an edge (a, b) represents a win of a over b.

Notation and Definitions. Unless noted otherwise, all graphs in the chapter are tournament graphs over n vertices, where n is a power of 2, and all SE tournaments are balanced. In Table 8.2, we define the notation that will be used in the rest of this chapter. For the definitions, let $\mathcal{A} \in V$ be any node, let $X, Y \subseteq V$ be such that $X \cap Y = \emptyset$.

Consider a tournament graph $G = (V, E)$. We say that $\mathcal{A} \in V$ is a king over another

Table 8.1: Notation

$N^{out}(\mathcal{A}) = \{v \mid (\mathcal{A}, v) \in E\}$ $N^{out}_X(\mathcal{A}) = N^{out}(\mathcal{A}) \cap X$
$N^{in}(\mathcal{A}) = \{v \mid (v, \mathcal{A}) \in E\}$ $N^{in}_X(\mathcal{A}) = N^{in}(\mathcal{A}) \cap X$
$out(\mathcal{A}) = \lvert N^{out}(\mathcal{A})\rvert$ $out_X(\mathcal{A}) = \lvert N^{out}_X(\mathcal{A})\rvert$
$in(\mathcal{A}) = \lvert N^{in}(\mathcal{A})\rvert$ $in_X(\mathcal{A}) = \lvert N^{in}_X(\mathcal{A})\rvert$
$\mathcal{H}^{in}(\mathcal{A}) = \{v \mid v \in N^{in}(\mathcal{A}), out(v) > out(\mathcal{A})\}$ $\mathcal{H}^{out}(\mathcal{A}) = \{v \mid v \in N^{out}(\mathcal{A}), out(v) > out(\mathcal{A})\}$
$\mathcal{H}(\mathcal{A}) = \mathcal{H}^{in}(\mathcal{A}) \cup \mathcal{H}^{out}(\mathcal{A})$
$E(X, Y) = \{(u, v) \mid (u, v) \in E, u \in X, v \in Y\}$
$\mathcal{M}(X, Y)$ is a maximal matching from X to Y $\mathcal{CM}(X, Y)$ is the canonical matching (Section 8.3.1.1)

Table 8.2: A summary of the notation used in this chapter.

node $x \in V$ if either $(\mathcal{A}, x) \in E$ or there exists $y \in V$ such that $(\mathcal{A}, y), (y, x) \in E$. A *king* in G is a node $\mathcal{A}$ which is a king over all $x \in V \setminus \{\mathcal{A}\}$. We say that set S covers a set T if for every $t \in T$ there is some $s \in S$ so that $(s, t) \in E$. Thus $N^{out}(\mathcal{A})$ covers the graph if and only if $\mathcal{A}$ is a king.

If one can efficiently construct a winning SE tournament bracket for a player $\mathcal{A}$, we say that $\mathcal{A}$ is an *SE winner*. We use the ranking Π formed by sorting the players in nondecreasing order of their outdegree.

We will construct SE tournaments as a series of matchings where each successive one will be over the winners of the previous one. A matching is defined as a set of pairs of vertices where each vertex appears in at most one pair. In our setting, these pairs are directed, so a matching from X to Y will consist only of edges that are directed from X to Y. If an edge is directed from x to y, then we refer to x as a *source*. Further, given a matching M from the sets X to Y, we will use the notation $X \setminus M$ to refer to the vertices in X that are not contained in the matching. A perfect matching from X to Y is one where every vertex of X is matched with a vertex of Y and $|X| = |Y|$. A perfect matching in a set S is a perfect matching from some $S' \subseteq S$ to $S \setminus S'$.

8.1 Motivation and Counterexamples

We will now discuss the motivation for our assumptions on the graph. We will look at some necessary and sufficient conditions for the top K players to win an SE tournament. We begin with an example.

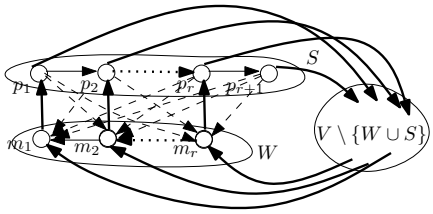

Figure 8.1: p_i only loses to m_i and p_j for $j < i$. No matter how the other edges of the tournament graph are placed, since the p_i beat everyone else and the m_i lose to everyone else, all SE tournament winners are in S.

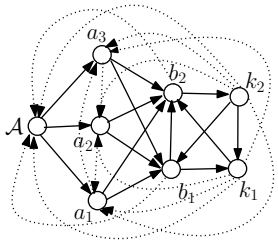

Figure 8.2: Example in which the two highest outdegree nodes, k_1 and k_2, have a matching into them but $\mathcal{A}$ cannot win an SE tournament.

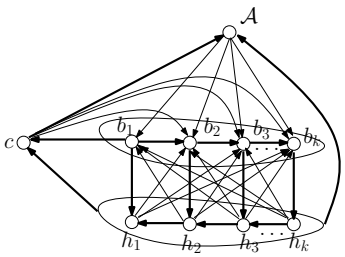

Figure 8.3: Example where there is a matching from $N^{out}(\mathcal{A})$ onto the k highest degree nodes but $\mathcal{A}$ can't win an SE tournament.

Consider the transitive tournament graph G with nodes $v_1, \ldots, v_n$, where v_i beats all nodes v_j for $j > i$. Then v_1 is the winner of all SE tournaments on G. Now, take any perfect matching from $\{v_1, \ldots, v_{n/2}\}$ to $\{v_{n/2+1}, \ldots, v_n\}$ and reverse these edges to create a *back-matching*. This gives each node from the weaker half of G a win against some node from the stronger half. The new outdegree ranking only swaps $v_{n/2}$ and $v_{n/2+1}$, however now the top $n/2-1$ players are SE winners: each of these nodes still beats at least $n/2$ other players, and the back-edges of the matching also make each one also a king. Prior work showed that this condition is sufficient for these players to be an SE tournament winner [143]. Thus, adding a back-matching to a transitive tournament can dramatically increase the set of winners. Our goal is to understand the impact of such back-edge matchings in general tournaments. As a warm-up, we consider the nodes of second and third highest outdegree. By case analysis, one can show the following theorems.

Theorem 14. *Let $G = (V, E)$ be a tournament graph and let a be the node of second highest outdegree. Then a single-elimination tournament bracket can be fixed for a if and only if there is no Condorcet winner in G.*

Proof sketch. Let m be the node with the highest outdegree, a the second highest, and x is a node that beats m. The proof proceeds in 3 cases as demonstrated by Figure 8.4.

The first is that $x = a$. Here, a is a king that beats all nodes of higher outdegree. By [143], a can win a single-elimination tournament. The second case is that a beats x. Here, a is also a king. It can be shown that there exists a perfect matching that includes (x, m), a is among the sources of the matching and a has a maximal outdegree among the sources. Again, by [143], a can win a single-elimination tournament. The final case is that x beats a. Here, we note that since x's outdegree is no greater than a and beats both m and a, it must be beaten by two nodes, v_1 and v_2, that a also beats. In this case, there always exists a matching such that v_1 or v_2 is a source, a is a king over the sources, and a beats at least $n/4$ of the sources. □

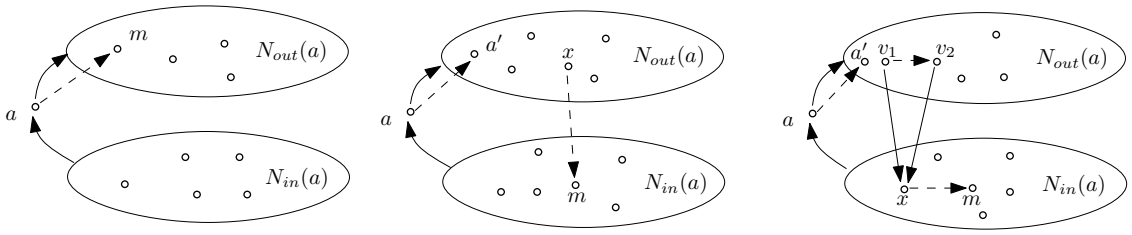

Figure 8.4: The three cases for the second strongest player, a where m is the strongest player.

We can generalize this theorem into the following result that for the top 3 nodes in the graph.

Theorem 15. ***(Top first, second and third node)*** *Let a be such that $|\mathcal{H}(a)| = K$ in a tournament $G = (V, E)$. Suppose there is a matching from $V \setminus \mathcal{H}(a)$ onto $\mathcal{H}(a)$. If $K \leq 1$, then for all n, a can win a single-elimination tournament. If $K = 2$, then for all $n \geq 16$, a can win a single-elimination tournament.*

The proof of this theorem is very similar to that of Theorem 14. The only change is that for the case of the third largest node, the two largest, m_1 and m_2 are beaten by some nodes a_1 and a_2. Since there is an edge between them (assume a_1a_2 without loss of generality), a_1 must be either in $N^{out}(a)$ or beaten by 3 nodes in that set. Similarly, a_2 must be in $N^{out}(a)$ or beaten by 2 not necessarily distinct nodes. Simple case analysis shows that no matter the overlap of these extra nodes, there exists a matching that ensures a is a king over a_1 and a_2 (and all other nodes) in round 2 of the tournament.

Proof. The cases where $K = 0$ and $K = 1$ are covered by Theorem 14 so we will focus only on the case where $K = 2$, i.e. $\mathcal{H}(a) = \{m_1, m_2\}$. If $m_1 \in N^{out}(a)$, or $m_2 \in N^{out}(a)$ then the proof when $K = 1$ works. Therefore, we will assume that both m_1 and m_2 are in $N^{in}(a)$. By the Theorem statement, let the matching be $\{(x_1, m_1), (x_2, m_2)\}$.

Suppose first that $x_1, x_2 \in N^{out}(a)$. The obvious thing to do is to match x_1 to m_1 and x_2 to m_2 and then complete the matching as in the canonical matching. This works provided $out(a) > 2$ so that a can be matched to some node that isn't x_1 or x_2. This case can be avoided if $n \geq 8$ and $in(a) \geq 5$. This guarantees that there are at least 3 nodes other than m_1, m_2 in $N^{in}(a)$. The number of edges into those two nodes from $\{x_1, x_2\}$ is at least 3 and so one of x_1 or x_2 must have outdegree at least 3. 3 would be larger than $out(a)$ which is a contradiction to x_1 and x_2 not being m_1 or m_2.

Now consider the case when $x_1 \in N^{out}(a)$ and $x_2 \in N^{in}(a)$. Since x_2 is of lower outdegree than a, x_2 must have indegree at least 2 since it beats a and m_2. Let $z_1, z_2 \in N^{out}(a)$ both beat x_2. Let M' be a maximal matching from $N^{out}(a) \setminus \{x_1\}$ into $N^{in}(a) \setminus \{m_1, x_2, m_2\}$. If $M' \cup \{x_1\}$ contains either z_i for one of $i \in \{1, 2\}$, then we can pick any $a' \neq z_i, x_1$ to match to a. Just as in the canonical matching, if we unmatch a' from M', then the number of unmatched nodes from $N^{in}(a)$ is even. Therefore, the survivors both lose at most one

inneighbor from $N^{out}(a)$ and at least one outneighbor from $N^{in}(a)$. If neither $z_1 \in M' \cup \{x_1\}$, nor $z_2 \in M' \cup \{x_1\}$, then match them to each other. In this case there is a counterexample on 8 nodes:

- $a : x_1, z_1, z_2,$
- $x_1 : m_1, x_2, y,$
- $z_1 : z_2, x_2, x_1,$
- $z_2 : x_1, x_2,$
- $y : z_1, a,$
- $x_2 : y, m_2, a,$
- $m_1 : y, x_2, z_1, z_2, a,$
- $m_2 : m_1, y, x_1, z_1, z_2, a.$

In order for a to win in this counterexample, x_2 must be matched to some node of $N^{out}(a)$ but then m_2 cannot lose.

Given that $n = 8$ doesn't necessarily work, assume $n \geq 16$. Now, $out(a) \geq \lceil (n-3)/2 \rceil = (n-2)/2 \geq 7$. We can extend $M' \cup \{(x_1, m_1), (x_2, m_2)\}$ just as in the canonical matching - since the outdegree of a is high enough, there is at least one node to match a' to. If $out(a) = l + n/2 - 1$, then

$$in(a) - 3 = n - 1 - 3 - (l + n/2 - 1) = n/2 - l - 3 \geq 5 - l.$$

If $5 - l \geq 2$, then $|M'| \geq 1$, and the number of nodes remaining after the first round is at least $\lfloor ((n-2)/2 - 1 + 2)/2 \rfloor = n/4$. Otherwise, $l \geq 4$, and $out(a) \geq n/2 + 3$. Then the number of nodes remaining after the first round is at least $\lfloor (n/2 + 3 - 1 + 1)/2 \rfloor \geq n/4$. Since a is a king it can win the tournament.

The final case is when $x_1, x_2 \in N^{in}(a)$. Without loss of generality, also let x_1 beat x_2. Here there is a counterexample for $n = 8$ as demonstrated in Figure 8.2.

Let $n \geq 16$, which implies that $out(a) \geq (n-2)/2 \geq (16-2)/2 = 7$. There are at least 3 inneighbors of x_1 and at least two inneighbors of x_2 in $N^{out}(a)$. We create a maximal matching M' from $N^{out}(a)$ into $N^{in}(a) \setminus \{x_1, x_2, m_1, m_2\}$. If for either x_1 or x_2 none of their inneighbors are sources of M' then there is a matching on their inneighbors in $N^{out}(a)$ so that the matching sources contain at least one inneighbor for each x_i. One can finish the matching just as in the canonical case. a can be matched since $out(a) > 4$. a will be a king in the remaining tournament. It remains to show that it has outdegree at least $n/4$. Let $out(a) = l + (n-2)/2 = n/2 + l - 1$. Then

$$|N^{in}(a) \setminus \{x_1, x_2, m_1, m_2\}| = n - 5 - n/2 - l + 1 = n/2 - 4 - l,$$

and

$$|M'| \geq n/4 - 2 - l/2.$$

The number of surviving outneighbors of a is

$$\lfloor (n/2 + l - 1 - 1 + n/2 - 2 - l/2)/2 \rfloor \geq n/2 - 2 \geq n/4$$

as desired. □

In Figure 8.1 we give an example of a tournament and a subset S consisting of the top $r+1$ outdegree nodes such that there is a matching of size r from a subset $W = \{m_1, \ldots, m_r\}$ of $V \setminus S$ into S, but no matching of size $r+1$ from $V \setminus S$ into S. Figure 8.1 only shows some of the graph edges. The edges within $V \setminus (W \cup S)$ are arbitrary, all nodes of S beat all nodes of $V \setminus (W \cup S)$, and all nodes of W lose to all nodes of $V \setminus (W \cup S)$. We can show that any node $\mathcal{A} \notin S$ cannot be an SE winner. p_1 only loses to m_1 and m_1 loses to everyone else so p_1 must be matched with m_1 in the first round if it is to ever be eliminated. Similarly, for any $i \leq r$, each p_i must be matched with m_i in the first round. Since all of the nodes that could possibly beat p_{r+1} lose in the first round, no one is left to beat p_{r+1} and $\mathcal{A}$ cannot win. Therefore, the only possible SE winners are contained in S. We have shown that for any r there exists a graph in which there is no matching onto the top r outdegree nodes and the $(r+1)$st outdegree node is not an SE winner. From this, we can conclude that the existence of a perfect matching from $V \setminus \mathcal{H}(\mathcal{A})$ into $\mathcal{H}(\mathcal{A})$ is, in a sense, necessary, in order for a node $\mathcal{A}$ to be an SE winner.

Now suppose that there is a perfect matching in G from $V \setminus \mathcal{H}(\mathcal{A})$ onto $\mathcal{H}(\mathcal{A})$. Can we conclude that the bracket can be fixed for $\mathcal{A}$? This turns out not to be true. Consider Figure 8.2. Here $\mathcal{H}(\mathcal{A})$ consists only of k_1 and k_2. These nodes are only beaten by b_1 and b_2 respectively, who lose to every other player except $\mathcal{A}$, so b_i and k_i must be matched in round 1. The a_i are symmetric, so without loss of generality we can match $\mathcal{A}$ to a_1 in round 1. The two remaining nodes, a_2 and a_3, must also be matched. After round 1 the nodes that survive are $\mathcal{A}, a_3, b_1, b_2$. However, $\mathcal{A}$ needs to have outdegree at least 2 to survive the next two rounds. As it only has outdegree 1, $\mathcal{A}$ cannot win an SE tournament.

A similar problem can arise when the matching comes from $N^{out}(\mathcal{A})$ instead of $N^{in}(\mathcal{A})$. Figure 8.3 gives an example of a graph construction for any $n \geq 8$ for which the node ranked $n/2$ cannot win any SE tournament even though there is a matching onto $\mathcal{H}(\mathcal{A}) = \cup_{i=1}^{k} h_i$. Each h_i only loses to b_i and $\cup_{j>i} h_j$. Each b_i only beats $\cup_{j>i} b_j$, except for b_1 who also beats c. The problem arises with who to match $\mathcal{A}$ to in the first round so that it can win the match. By induction, one can argue that every h_i for $i > 1$ must be matched to b_i in round 1. $\mathcal{A}$ must be matched to some node in $N^{out}(\mathcal{A})$, but only b_1 remains unmatched. This leaves h_1 and c who must be matched as well. However, in round 2, all nodes that beat h_1 have been eliminated and it is now a Condorcet winner in the induced subgraph. Therefore, it must be the winner of any SE tournament.

A common issue in the above counterexamples is that $\mathcal{H}(\mathcal{A})$ is too large while $out(\mathcal{A})$ is too small. Another commonality is that $\mathcal{H}(\mathcal{A}) = \mathcal{H}^{in}(\mathcal{A})$. Hence a better condition to look for is a matching from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ onto $\mathcal{H}^{in}(\mathcal{A})$, and not necessarily onto $\mathcal{H}(\mathcal{A})$.

Finally, a natural question is, how reasonable is the assumption of the existence of a matching from lower ranked players to higher ranked players? Consider the Braverman-Mossel model [32] for generating tournament graphs. In this model, one assumes an underlying ranking $v_1 \cdots v_n$ of the players according to skill. The tournament is generated by adding an edge (v_i, v_j) with probability p if $j < i$ and $1 - p$ if $j > i$ for $p < \frac{1}{2}$. This model can be viewed as a transitive tournament with each edge reversed with probability p. A classic result of [53] is that a bipartite graph with n nodes on each side with $2n \ln n$ edges selected uniformly at random contains a perfect matching with high probability. If a graph is generated by the Braverman-Mossel model with $p > \frac{4 \ln n}{n}$, then we expect there to be $n \ln n$ back edges from $v_{n/2} \cdots v_n$ to $v_1 \cdots v_{n/2-1}$. Therefore, in almost all such tournaments, a backedge matching exists.

8.2 Main Results

We are now ready to introduce our main result. As the proof is quite technical, we will first provide an intuitive sketch, some of the necessary Lemmas, and a more detailed account of the key part of our proof. All full proofs are included in the Appendix.

We present two main results. The first generalizes the idea of a king, and shows that if a node $\mathcal{A}$ is a king except for some subset and $\mathcal{A}$ beats many nodes that beat a king of that subset, then $\mathcal{A}$ is an SE winner.

Lemma 13 (**Kings Except for a T subset**). *Let $\mathcal{A}$ be a node in a tournament G and let T be a subset of $N^{in}(\mathcal{A})$ of size $|T| = 2^k$ for some k. Suppose that $\mathcal{A}$ is a king in $G \setminus T$ and $|N^{out}(\mathcal{A})| \geq |N^{in}(\mathcal{A})|$. Let t be a king in T with outdegree in T at least $\lfloor |T|/2 \rfloor$. Suppose that $|N^{in}(t) \cap N^{out}(\mathcal{A})| \geq |T|$. Then $\mathcal{A}$ is an SE winner.*

The key observation in proving Lemma 13 is that t can win an SE tournament over just the subgraph consisting of T in $\log |T|$ rounds. At the same time, there are at least $|T|$ nodes in $N^{out}(\mathcal{A})$ that beat t. In the worst case, these cannot eliminate any other nodes in $N^{in}(\mathcal{A})$ so they must be matched against each other for $\log |T|$ rounds as well. However, given the size, we are guaranteed that at least 1 will survive to eliminate t. At this point, $\mathcal{A}$ will be a king of high outdegree over the induced subgraph. The technical details of the proof proceed by induction on the size of T. Lemma 13 is used in the proof of our main theorem below. We highlight its use in the intuitive sketch.

We now address the main question of this chapter - what can we show when a matching from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ to $\mathcal{H}^{in}(\mathcal{A})$ exists?

Theorem 16 (**Not a King but Matching into $\mathcal{H}^{in}(\mathcal{A})$**). *There exists a constant n_0 such that for all $n \geq n_0$ the following holds. Let $G = (V, E)$ be a tournament graph on n nodes, $\mathcal{A} \in V$. Suppose there is a matching M from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ onto $\mathcal{H}^{in}(\mathcal{A})$ of size K. If $K \leq (n - 6)/7$, then $\mathcal{A}$ is an SE winner.*

The key restriction in this Theorem concerns the number of higher ranked players who beat a player, not the actual rank of that player. However, we are able to apply the fact that a player of rank k has outdegree at least $(n-k-1)/2$ to obtain a nice corollary for large tournament graphs: Any one of the top 19% of the nodes are SE winners, provided there is a matching onto the nodes of higher outdegree.

Corollary 3. *There exists a constant n_0 so that for all tournaments G on $n > n_0$ nodes the following holds. Let $\mathcal{A}$ be among the top $(6n+7)/31 \geq .19n$ highest outdegree nodes. If there is a matching from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ onto $\mathcal{H}^{in}(\mathcal{A})$, then $\mathcal{A}$ is an SE winner.*

8.2.1 Intuition

We now give an intuitive sketch about how one might go about proving Theorem 16. The overall strategy of our proof is to set up the first round of the SE tournament, so that all of the high outdegree nodes that beat $\mathcal{A}$ are eliminated, and in the remaining tournament, $\mathcal{A}$ is a king over almost the entire graph, so that Lemma 13 can be applied.

At first glance, one might try to build the first round by using the existing matching, M, from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ to $\mathcal{H}^{in}(\mathcal{A})$ and then finding some maximal matching M' from $N^{out}(\mathcal{A}) \setminus M$ to $N^{in}(\mathcal{A}) \setminus M$. The matching M' will guarantee that as many elements as possible of $N^{out}(\mathcal{A})$ will survive to compete in the second round. To complete round 1, the potentially remaining nodes in $N^{out}(\mathcal{A}) \setminus (M \cup M')$ should be matched amongst themselves, and the same for $N^{in}(\mathcal{A}) \setminus (M \cup M')$ in a matching called M''.

$\mathcal{A}$ is initially a king in G over any node with no larger outdegree than it (i.e. $V \setminus \mathcal{H}^{in}(\mathcal{A})$). However, if we do not create the matching M'' above carefully $\mathcal{A}$ may no longer be a king over the sources of M. Even worse, some source of M may lose all of the nodes that can potentially beat, and might become a Condorcet winner in the graph induced by the winners of round 1. This is demonstrated in Figure 8.5. In this example, we would like to fix the bracket for P_2, the second strongest player. P_3 can beat P_1, but only P_{n-1} and P_n beat P_3. If we use any matching of $N^{out}(P_2)$ that does not match P_n with P_{n-1}, P_3 will be a Condorcet winner in round 2, and P_2 cannot win.

The failure of this example motivates our approach. We begin our construction of round 1 as before. We use the perfect matching M from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ to $\mathcal{H}^{in}(\mathcal{A})$ and M', a maximal matching M' from $N^{out}(\mathcal{A}) \setminus M$ to $N^{in}(\mathcal{A}) \setminus M$. At this point, we want to guarantee that as many of the sources of M as possible are still covered by winners of round 1. We start by finding the set T of sources of M that are not currently beaten by some source in M', or by $\mathcal{A}$. Because these nodes are all of lower outdegree than $\mathcal{A}$, we can argue that there is some subset S which is a subset of $N^{out}(\mathcal{A}) \setminus (M \cup M')$ that covers T. We use a greedy approach (Algorithm 8) to match up the nodes of S in round 1 so that the winners of this matching cover as many nodes of T as possible. We are able to show (in Lemma 14) that the set U of nodes of T that are not covered by the first round winners from S is very small: it has size at most $O(\sqrt{|T|})$. This will allow us to show that we can eliminate U in later rounds.

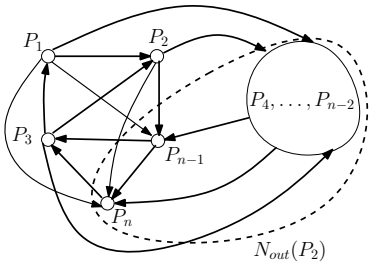

Figure 8.5: An example where an arbitrary matching of $N^{out}(P_2)$ is likely to fail.

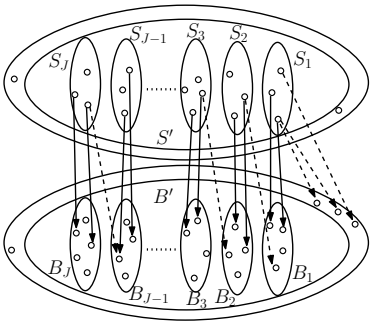

Figure 8.6: The construction of the sets S_i and B_i in Theorem 17.

We design the next rounds using Lemma 13. To do this, we use the largest outdegree node t in T and find a set P, the size of which is a power of 2, that contains both t and U. The final requirements of Lemma 13 are that $\mathcal{A}$ beats at least as many first round winners outside P as it loses to (which we show using Theorem 17) and that the number of nodes from $N^{out}(A)$ that beat t and survive round 1 is at least $|P|$. To fulfill this last requirement, we add an extra iteration (for $q = 1$) in Algorithm 8 which constructs the first round matching of S so that enough nodes that beat t survive round 1.

Summary. We create the first round of the tournament by using M, a maximal matching M' from the remaining nodes of $N^{out}(\mathcal{A})$ to the remaining nodes of $N^{in}(\mathcal{A})$, and a greedily selected perfect matching M'' on S. Many sources of M'' beat t, and almost all of T is covered by the sources of M''. This does not fully specify the first round matching. A few nodes may remain unmatched, specifically $N^{in}(\mathcal{A}) \setminus (M \cup M')$, $N^{out}(\mathcal{A}) \setminus (M \cup M' \cup M'')$ and $\mathcal{A}$ itself. The final details are included in the proof sketch at the end of this section. The goal of the first round matching is to ensure that the requirements of Lemma 13 are met and the remaining rounds of the tournament can be completed so that $\mathcal{A}$ wins.

8.2.2 Technical details.

With the above overview of the proof technique, we now introduce the necessary lemmas. As an SE tournament is a series of $\log n$ matchings, these lemmas are about the existence of matchings with desirable properties. The first is a very general result that can be specifically applied to lower-bound how large a matching can be found from $N^{out}(\mathcal{A})$ to $N^{in}(\mathcal{A}) \setminus \mathcal{H}^{in}(\mathcal{A})$.

Theorem 17 (**Large Matching**). *Let $h \in \mathbb{Z}$, possibly negative. Let S and B be disjoint sets such that $\forall X \subset B$, $|E(S, X)| \geq \binom{|X|}{2} - h|X|$. Then there exists a matching between S and B of size at least $\frac{|B|-2h-1}{2}$.*

Proof. Recall that M is a maximal matching from a set S to a set B if and only if there are no augmenting paths from the unmatched elements of S to the unmatched element of B.

Algorithm 8 Greedy Matching

Input: $G = (V, E)$ a tournament and $S, T \subset V$, $t \in V$; Output: Matching M
Let $A_1 = N_S^{in}(t)$, $U_1 = T$, $i = 1$, $L_0 = \emptyset$, $M = \emptyset$.
for $q = 1, 2$ **do**
  while $|A_i| \geq 2$ **do**
    Let $x_i, y_i \in A_i$ have larger outdegree to U_i than all the other elements in A_i; x_i beats y_i.
    $M \leftarrow M \cup \{(x_i, y_i)\}$
    $L_i \leftarrow L_{i-1} \cup \{y_i\}$
    $U_{i+1} \leftarrow U_i \setminus N^{out}(x_i)$
    $A_{i+1} = \cup_{v \in U_{i+1}} N_{A_i}^{in}(v) \setminus L_i$
    $i \leftarrow i + 1$
  $A_i = \cup_{v \in U_i} N_S^{in}(U_i)$

Our proof will proceed by using the large number of edges from S to any subset X of B to lower-bound the size of the matching.

Let M be a maximal matching from S to B. We refer to the sources of M as S' and the sinks as B'. We iteratively build up a family of sets S_j and B_j that consist of augmenting paths from the unmatched nodes in B.

Let S_1 be the subset of S' which contains all nodes with edges to $B \setminus B'$. Let B_1 be the nodes matched to S_1 by M. Now, we inductively define S_j as the nodes in $S' \setminus \cup_{i=1}^{j-1} S_i$ that have edges to B_{j-1}, where B_{j-1} are the nodes matched to S_{j-1} by M.

This process can be repeated up to some index $J + 1$ such that there are no more nodes in $S' \setminus \cup_{i=1}^{J} S_i$ with edges to B_J. Let $\bar{S} = \cup_{i \leq J} S_i$ and $\bar{B} = (B \setminus B') \cup (\cup_{i \leq J} B_i)$.

First, note that there are no edges from $S \setminus S'$ to $\bar{B}$ since M is maximal. If there were, we would have an augmenting path. Therefore, all edges into $\bar{B}$ come from $\bar{S}$. The number of edges from $\bar{S}$ into $\bar{B}$ is at most $|\bar{B}||\bar{S}|$ (the number of edges in a complete bipartite graph) and at least $|\bar{B}|(|\bar{B}| - 1 - 2h)/2$ by the Theorem statement. Thus, we can conclude that

$$|M| = |B \setminus \bar{B}| + |\bar{S}| \geq$$

$$(|B| - |\bar{B}|) + \frac{(|\bar{B}| - 1 - 2h)}{2} \geq \frac{(|B| - 1 - 2h)}{2}.$$

□

Theorem 17 is used in the proof of Theorem 16 to argue about a lower bound on the size of $N^{out}(\mathcal{A})$ after the first round. An example application of this theorem is to set S to $N^{out}(\mathcal{A})$ and B to $N^{in}(\mathcal{A}) \setminus (M \cup \mathcal{H}^{in}(\mathcal{A}))$. Here, the conditions of the Theorem are met: we can show that for every subset X, $E(S, X) \geq \binom{|X|}{2} + |X|$ because every vertex in B beats $\mathcal{A}$ and is of lower outdegree than $\mathcal{A}$.

The other very important part of our proof is Algorithm 8. As mentioned earlier, it is a greedy way of creating a matching in a set S such that the sources cover many elements in a

set T. It iteratively finds the source in S that covers the most uncovered elements of T and matches it with a vertex that it beats. The first iteration of the loop deals with an element t that is a king over T. This loop only considers the subset of S that beats t and guarantees that at least half of the nodes that beat t in S are preserved as sources. At any time in the algorithm, U_i is the set of the nodes that are currently not covered by the sources of the matching M, A_i is the set of sources that beat any element in U_i, and L_i is the set of nodes that lose in M and are excluded from A_i.

We want to lower-bound the size of the generated cover. The main idea of the proof is that we initially have many edges from S to T, and specifically at least $\binom{|X|+1}{2}$ to each $X \subseteq T$. If we consider the first pair (x_1, y_1) added to M, then we can say x_1 covers k elements of T. Therefore, we now need to cover only a subset of size $|T| - k$ which has at least $\binom{|T|-k}{2}$ edges into it. However, this may include edges from y_1. When we remove y_1, we may lose up to $|T|$ edges. The key observation is that for the pair (x_2, y_2), y_2's outdegree is upper-bounded by x_1 so we are able to bound the number of edges lost by the matching as the number of vertices currently covered plus $|T|$. We then show that there will always be enough edges and sources to increase the size of the matching until at most $2\sqrt{|T|} + 1$ nodes remain uncovered.

Lemma 14. *Let $G = (V, E)$ be a tournament graph. Let $S \subseteq V$ and $T \subseteq V$ be disjoint sets such that for all $X \subseteq T$, the number of edges from S to X is at least $\binom{|X|+1}{2}$. Let $t \in V$ be given. Algorithm 8 generates a matching, M, in S such that at least $|T| - 1 - 2\sqrt{|T|}$ nodes in T are beaten by at least one source in M and at least $(in_S(t) - 2)/2$ of the sources also beat t.*

Proof. We need to define some additional concepts for the proof. The first is the set of covered nodes at iteration i, C_i, where $C_1 = \emptyset$. C_i is exactly $T \setminus U_i$ (so $|T| = |C_i| + |U_i|$). Let $d_i = |C_{i+1}| - |C_i|$ be the number of new nodes covered by iteration i. Our goal is to lower-bound the size of $|C_i|$ when the algorithm quits.

Consider the first execution of the WHILE loop. Let i_0 be the iteration at which the loop exits. This loop greedily covered T but only used vertices that also beat t. We will lower-bound the number of edges that remain from all unmatched sources in S (the set A_{i_0}) to U_{i_0}. At this point, $|C_{i_0}| = \sum_{j=1}^{i_0} d_j$. The number of edges from L_{i_0} to U_{i_0} is at most $|T| - |C_{i_0}| + \sum_{j=1}^{i_0-1} d_j \leq |T|$ since we picked the nodes so that $out_{U_i}(y_i) \leq out_{U_{i-1}}(x_{i-1}) = d_{i-1}$, and $out_{U_{i_0}}(y_1) \leq |U_{i_0}|$. Thus we can obtain a lower bound on the number of edges between A_{i_0} and U_{i_0}: $|E(A_{i_0}, U_{i_0})| \geq \binom{|U_{i_0}|+1}{2} - |T|$.

Let $j > i_0$ be any round in the second WHILE loop. As above, $|C_j| = |C_{i_0}| + \sum_{k=i_0+1}^{j} d_k$ and the number of edges from L_j to U_j is at most

$$|U_j| + |T| + \sum_{k=i_0+1}^{j-1} d_k = 2|T| - |C_j| + |C_j| - |C_{i_0}| \leq 2|T|.$$

We can lower-bound the number of usable edges from A_j to U_j as

$$|E(A_j, U_j)| \geq \binom{|U_j|+1}{2} - 2|T| \geq$$

$$(|T|^2 + |C_j|^2 - (2|T|+1)|C_j| - 3|T|)/2.$$

The second WHILE loop exits when $|A_j| \leq 1$. Therefore, when the algorithm finishes, $|A_j| \leq 1$ and $|E(A_j, U_j)| \leq |U_j| = |T| - |C_j|$. We have:

$$(|T|^2 + |C_j|^2 - (2|T|+1)|C_j| - 3|T|)/2 \leq |T| - |C_j|,$$

This can be simplified as follows.

$$|C_j|^2 - (2|T|-1)|C_j| - 5|T| + |T|^2 \leq 0.$$

$$|C_j| \geq |T| - 1/2 - \sqrt{|T|^2 - |T| + 1/4 + 5|T| - |T|^2} =$$
$$|T| - 1/2 - \sqrt{4|T| + 1/4} \geq |T| - 1 - 2\sqrt{|T|}.$$

That is, the number of covered nodes is at least $|T| - 1 - 2\sqrt{|T|}$. After round i_0 we have at least i_0 sources in M covering t and at least $in_S(t) - 2i_0 - 1$ nodes of $N_S^{in}(t)$ that were not used in creating the rest of the matching because they did not cover any element of U_{i_0}. Match these among themselves to obtain at least $i_0 + \lfloor (in_S(t) - 1 - 2i_0)/2 \rfloor \geq (in_S(t) - 2)/2$ sources of the matching that are inneighbors of t. Complete the matching M from S to S by matching the rest of the nodes of S arbitrarily. □

The bounds on the greedy matching algorithm are only positive if $|T| > 5$. We don't want our bounds in Theorem 16 to depend on the size of the matching into $\mathcal{H}^{in}(\mathcal{A})$. We now present a sketch of the proof that ignores this difficulty. The full proof contained in the Appendix fixes this problem through the introduction of a technical Lemma, Lemma 18. This lemma allows one to artificially boost the size of T to guarantee that the above process will always work. Additionally, this proof sketch assumes that that the indegree of node $\mathcal{A}$ coming from the sources of M is large enough. This assumption is also lifted in the Appendix.

Proof sketch of Theorem 16: This proof proceeds by constructing the first round matching in stages. First, we will use M, the matching given by the theorem statement, and construct M', a maximal matching. Next, we show how to match $\mathcal{A}$ and construct the covering of the sources of M using Algorithm 8. Finally, we argue that the constructed first round matching satisfies the requirements of Lemma 13.

For simplicity, let $A = N^{out}(\mathcal{A})$ and $B = N^{in}(\mathcal{A})$. We divide the sources of M onto $\mathcal{H}^{in}(\mathcal{A})$ into two sets, A_T and B_T, where A_T are the sources of M in A while B_T are the sources in B. We can also divide $\mathcal{H}^{in}(\mathcal{A})$ into two sets, H_1 and H_2, where H_1 are the nodes matched to A_T and H_2 are matched to B_T by M. In order to later argue about the size of

matchings, let $|A_T| = |H_1| = h$ and $|B_T| = |H_2| = k$. This means that K, the size of M is exactly $k + h$.

Let $B_{\text{rest}} = B \setminus (B_T \cup \mathcal{H}^{in}(\mathcal{A}))$ be the nodes who beat $\mathcal{A}$ and are not part of M. Take M' to be any maximal matching from $A \setminus A_T$ to B_{rest}. We want to argue about the size of M' by using Theorem 17. First, note that $|B_{\text{rest}}| = |B| - k - K$. Now, since we removed A_T, of size h, we can only say that every node b in B_{rest} has at least $out_B(b) + 1 - h$ inneighbors from $A \setminus A_T$. Therefore, by Theorem 17,

$$|M'| \geq (|B| - K - k - 2h + 2 - 1)/2 = (|B| - 2K - h + 1)/2.$$

We will use this fact later when arguing about the outdegree of $\mathcal{A}$ after the first round.

Finally, note that B_{rest} consists only of lower ranked nodes than $\mathcal{A}$, so every node in B_{rest} has some source of M' or A_T as an inneighbor.

(Matching $\mathcal{A}$ to some node.) Consider the currently unmatched portion of A. Call this $A_{\text{rest}} = A \setminus (A_T \cup M')$. If there is some $a' \in A_{\text{rest}}$, then match $\mathcal{A}$ to a'. If A_{rest} is empty, then we can argue that $|M'| > 1$ since

$$|A \setminus A_T| = |A| - h \geq (n - K)/3 - h \geq (n - 4K)/3 > 1.$$

Since $M' > 1$, we can dislodge any edge (a', b') from M' and match $\mathcal{A}$ to a'. After removing a', the lower bound for $|M'|$ goes down by 1: $|M'| \geq (|B| - 2K - h - 1)/2$.

(Creating a matching of $A_{\text{rest}} \setminus \{a'\}$).) We now use Algorithm 8 to cover B_T. Let $S = A_{\text{rest}} \setminus \{a'\}$ and T be the subset of B_T consisting of the nodes that do not have inneighbors among the sources of M' and A_T. For simplicity in this proof we assume that $|T|$ and hence $|B_T|$ is large enough.

Every subset X of the nodes of T has at least $\binom{|X|}{2} + 2|X| - |X| = \binom{|X|}{2} + |X|$ inneighbors in S since each node in X can have lost at most one inneighbor, a'. Let $t \in B_T$ be the node with highest outdegree in B_T. Run Algorithm 8 on S, T, t. This outputs a matching M'' on the nodes of S that covers all of T except for a subset, U, of size at most $1 + 2\sqrt{|T|}$. There are also at least $in_S(t)/2 - 1$ sources of M'' that beat t. This completes the first round matching. Let G' be the graph induced by the surviving nodes.

(Handling U.) We will construct P, a subset of T, such that P contains U, and t is a king over P who beats at least half of P.

We selected t so that it is a king in T. Therefore, there is a subset of at most $|U|$ nodes in its outneighborhood in T that cover U. We can add these nodes together with enough other nodes of $N_T^{out}(t)$ to P so that $|P|$ is a power of 2 and t is a king in P that beats at least half of P. This is possible since U is very small compared to T.

We can assume that the size of P is 2^c where 2^c is the closest power of 2 greater than $3 + 4\sqrt{|T|}$, as we may need as many as $|U| \leq 1 + 2\sqrt{|T|}$ extra nodes added to P to guarantee that t is a king over P. We can further conclude that $|P| \leq 5 + 8\sqrt{|T|}$ since we can at most double $3 + 4\sqrt{|T|}$ to make $|P|$ be a power of 2.

From Algorithm 8 we know that at least

$$in_S(t)/2 - 1 \geq (|B_T| - 1)/4 - 1$$

inneighbors of t from S are in G'. Since we assumed that B_T is large enough, we have

$$(|B_T| - 1)/4 - 1 \geq 5 + 8\sqrt{|T|}.$$

Hence there exists a subset of the surviving nodes of $N_S^{in}(t)$ of size at least $|P|$. The requirements of Lemma 13 are satisfied if $out_{G'}(\mathcal{A}) \geq in_{G'}(\mathcal{A})$. We prove this below and thus show that $\mathcal{A}$ is an SE winner.

(Showing that $out_{G'}(\mathcal{A}) \geq in_{G'}(\mathcal{A})$.) The number of nodes of $N^{out}(\mathcal{A})$ that survive the first round is at least

$$\lfloor (|A| + |M'| + |A_T| - 1)/2 \rfloor.$$

The number of nodes of $N^{in}(\mathcal{A})$ that survive is at most $\lceil (|B| - |A_T| - |M'|)/2 \rceil$. It suffices to show that

$$|A| + |M'| + |A_T| - 1 \geq |B| - |A_T| - |M'|.$$

Recall that $|M'| \geq (|B| - 2K - h - 1)/2$ so we must only show that $|A| + |B| - 2K - h + 2h - 2 \geq |B|$, or that $|A| - 2K + h - 2 \geq 0$. Since $|A| \geq (n - K)/3$ it suffices to show that $(n - K) \geq 6K + 6$, or that $K \leq (n - 6)/7$, which is true by assumption. □

8.3 The Full Proof

8.3.1 Additional Tools

For the full proofs of the main results, we rely on a few constructions and facts that were not mentioned in the main part of the chapter. The first of these is the canonical matching. This is a generic matching construction that maximizes the surviving sources while minimizing the surviving sinks.

8.3.1.1 Canonical matching.

Let G be a tournament graph and let $A, B \subset V$ such that $|A| + |B|$ is even. A *canonical* matching, $\mathcal{CM}(A, B)$ is formed as follows: create a maximal matching M' from A to B. Match all of the nodes in A that are not in M' against each other, and all of the unmatched nodes in B against each other. If $|M'|$ is odd, then match the leftover node in A with the leftover node in B.

We now describe a canonical matching for a given king node $\mathcal{A}$. Let G be a tournament graph over an even number of nodes and $\mathcal{A}$ be a king in G with $out(\mathcal{A}) \geq in(\mathcal{A})$. In the following construction $\mathcal{CM}(\mathcal{A})$ we include $\mathcal{A}$ by modifying $\mathcal{CM}(N^{out}(\mathcal{A}), N^{in}(\mathcal{A}))$. Since $out(\mathcal{A}) \geq in(\mathcal{A})$ and n is even, $out(\mathcal{A})$ and $in(\mathcal{A})$ have different parity, and $out(\mathcal{A}) \geq 1 + in(\mathcal{A})$. Thus $|N^{out}(\mathcal{A}) \setminus M'| \geq 1$ and we can pick any node $a' \in N^{out}(\mathcal{A}) \setminus M'$ to match with $\mathcal{A}$. Match the nodes of $N^{out}(\mathcal{A}) \setminus M' \setminus \{a'\}$ amongst themselves. At most one node a'' is left over. Match the nodes of $N^{in}(\mathcal{A}) \setminus M'$ amongst themselves. At most one node b'' is left over, and it is left over iff a'' is. Match a'' and b'', completing $\mathcal{CM}(\mathcal{A})$. The proof of the following lemma follows from the maximality of M'.

Lemma 15. *Let $\mathcal{A}$ be a king such that $out(\mathcal{A}) \geq in(\mathcal{A})$. Let G' be the subtournament graph over the sources of $\mathcal{CM}(\mathcal{A})$. Then*

- *$\mathcal{A}$ is a king in G',*
- *$out_{G'}(\mathcal{A}) = \lfloor (out(\mathcal{A}) + |M'| - 1)/2 \rfloor$,*
- *$in_{G'}(\mathcal{A}) = \lfloor (in(\mathcal{A}) - |M'| + 1)/2 \rfloor$,*
- *$out_{G'}(\mathcal{A}) \geq in_{G'}(\mathcal{A})$.*

Proof. 1) follows since M' was maximal, and so any node in $N^{in}(a)$ that is not in M' must have some source of M' as an inneighbor. 2) follows since $|M'|$ of the nodes in $N^{out}(a)$ survive the matching M', one node is removed, and the rest are matched amongst themselves, possibly one losing to a node of $N^{in}(a)$. Hence, $out_{G'}(a) = |M'| + \lfloor (out(a) - 1 - |M'|)/2 \rfloor = \lfloor (out(a) - 1 + |M'|)/2$.

3) follows since $|M'|$ nodes of $N^{in}(a)$ are eliminated by M', and the rest are matched amongst themselves, possibly one beating a node of $N^{out}(a)$. Hence

$$in_{G'}(a) = \lceil (in(a) - |M'|)/2 \rceil = \lfloor (in(a) - |M'| + 1)/2 \rfloor.$$

4) If $|M'| \geq 1$, $in_{G'}(a) \leq \lfloor in(a)/2 \rfloor \leq out_{G'}(a)$. Otherwise, since a was a king and M' was maximal, $N^{in}(a) = \emptyset$. Then $out_{G'}(a) \geq 0 = in_{G'}(a)$. □

8.3.1.2 Bounds on $out(\mathcal{A})$.

We often need to argue about the size of a given set, given some constraints on the number of higher degree nodes that exist, or that a player beats. The following are useful facts of this type.

Fact 1. *For any tournament graph of size k, there exists a vertex with outdegree at least $\lfloor \frac{k}{2} \rfloor$.*

This follows directly from the fact that a tournament of size k has $\binom{k}{2}$ edges.

Lemma 16. *Let $\mathcal{A}$ be a node in a tournament graph $G = (V, E)$ with $|\mathcal{H}(\mathcal{A})| = k$. Then $out(\mathcal{A}) \geq \lfloor \frac{(n-k)}{2} \rfloor$.*

Proof. Let $|\mathcal{H}^{out}(a)| = k_1$, $|\mathcal{H}^{in}(a)| = k_2 = k - k_1$, and $out(a) = d$. Let $R = V \setminus \{a\} \setminus \mathcal{H}$. Then $|R \cap N^{in}(a)| = n - d - k_2 - 1$ and $|R| = n - k - 1$. Since for every $b \in R$, $out(b) \leq out(a) = d$, d is at least the average of the outdegrees of $R \cup \{a\}$ in G. The sum of these outdegrees is

$$\begin{aligned} & d + \tbinom{n-k-1}{2} + (n - d - k_2 - 1) + out_{\mathcal{H}}(R) \\ \geq \quad & (n-k-1)(n-k-2)/2 + (n-k) - 1 \\ = \quad & (n-k)(1 + (n-k-1)/2 - 1) + 1 - 1 \\ = \quad & (n-k)(n-k-1)/2 \end{aligned}$$

Since $|R \cup \{a\}| = n - k$ and d is integral, $d \geq \lfloor (n-k)/2 \rfloor$. □

Lemma 17. *Let $\mathcal{A}$ be a node in a tournament graph such that $|\mathcal{H}^{in}(\mathcal{A})| = k$. Then $out(\mathcal{A}) \geq (n-k)/3$.*

Proof. Let $A = N^{out}(a)$ and $B = N^{in}(a)$. The number of edges from A to $B \setminus \mathcal{H}^{in}(a)$ is at most $|A|(|B| - k)$ and least $\binom{|B|-k}{2} + |B| - k = (|B| - k)(|B| - k + 1)/2$ since for every $b \in B \setminus \mathcal{H}^{in}(a)$, $1 + out_B(b) \leq in_A(b)$. Hence,

$$|A| \geq (|B| - k + 1)/2 = (n - 1 - |A| - k + 1)/2 \implies |A| \geq (n-k)/3.$$

□

Fact 2. *Let x and y be nodes in a tournament graph such that $out(x) \geq out(y)$. Then the distance between x and y is at most 2. If $\mathcal{A}$ is a node such that for all $x \neq \mathcal{A}$, $out(\mathcal{A}) \geq out(x)$, then $\mathcal{A}$ is a king.*

8.3.1.3 Boosting set sizes

Our final additional tool is the technical lemma mentioned before the proof sketch of Theorem 16. Its application relies heavily on the greedy matching algorithm, Algorithm 8.

The bounds on the greedy matching algorithm given by Lemma 14 are only positive if $|T| > 5$. However, the way we will apply Lemma 14 in Theorem 16 will require that T be significantly larger than 5. We don't want our bounds in Theorem 16 to depend on the size of the matching into $\mathcal{H}^{in}(\mathcal{A})$, so we present the next lemma as a way of artificially boosting the size of T in order to guarantee that the above process will always work.

The intuition for the following technical lemma is that it is a method of picking a subset of nodes in $N^{out}(\mathcal{A})$, T, so that the requisite edges for the previous algorithm have no needed sources in T, and that $\forall X \subset T$, $|E(N^{out}(\mathcal{A}) \setminus T, X)| \geq \binom{|X|}{2} + 2|X|$.

Lemma 18. *Let C be a given constant. Let S and T be disjoint node sets of a tournament graph such that for every $t \in T$, $in_S(t) \geq out_T(t) + 2$, and $|T| < C$. Let $M \subseteq S$ such that $|S \setminus M| \geq (5C^2 + 17C + 4)/2$. Then there exists a subset $Z \subset S \setminus M$ such that $|Z| = 2(C - |T|)$ and $\forall Q \subseteq (Z \cup T)$, $|E(S \setminus Z, Q)| \geq \binom{|Q|}{2} + 2|Q|$.*

Proof. Form a subset $Y \subset S$ by including for every $t \in T$ exactly $out_T(t)+2$ of its inneighbors from S. We can lower bound the size of Y as $|Y| \leq \binom{|T|}{2} + 2|T| \leq C(C+3)/2$. These are the sources needed to apply Lemma 14 to the set T. Let $R = S \setminus (M \cup Y)$. Hence

$$|R| \geq (5C^2 + 17C + 4)/2 - C(C+3)/2 = 2C^2 + 7C + 2.$$

Now we can create the set Z. While $|Z| < 2(C - |T|)$: pick $z \in R$ of largest indegree and add z to Z while removing it from R. Additionally, remove from R exactly $C + 2$ of the inneighbors of z.

We now want to bound the number of edges removed from R. Notice that

$$|R| - (2C - 2|T| - 1)(C + 3) \geq 2C^2 + 7C + 2 + 2|T|(C + 3) - (2C^2 + 5C - 3) \geq 1 + 2(C + 2).$$

Since we have removed at most $(2C - 2|T| - 1)(C + 3)$ nodes from R, at each step the indegree of z is at least $(|R| - (2C - 2|T| - 1)(C + 3) - 1)/2 \geq C + 2$ by Fact 1.

Now consider $T \cup Z$. We will prove that $\forall Q \subseteq T \cup Z$, $|E(S \setminus Z, Q)| \geq \binom{|Q|}{2} + 2|Q|$ by induction on the number p of elements of Z contained in the subset $Q \subseteq T \cup Z$. The statement is clearly true when $p = 0$. Suppose it is true for all subsets with at most $p - 1$ elements of Z. Consider a subset Q with p elements of Z and let $z \in Q \cap Z$. Then we know by the induction hypothesis that $|E(S \setminus Z, Q \setminus \{z\})| \geq \binom{|Q \setminus \{z\}|}{2} + 2|Q \setminus \{z\}|$. Since $in_{S \setminus Z}(z) \geq C + 2 \geq |Q| + 2$, we can conclude that

$$|E(S \setminus Z, Q)| \geq \binom{|Q| - 1}{2} + 2(|Q| - 1) + |Q| + 2 = \binom{|Q|}{2} + 2|Q|.$$

□

8.3.2 Full Proofs of Main Results

Reminder of Lemma 13 [**Kings Except for a T subset**] Let $\mathcal{A}$ be a node in a tournament G and let T be a subset of $N^{in}(\mathcal{A})$ of size $|T| = 2^k$ for some k. Suppose that $\mathcal{A}$ is a king in $G \setminus T$ and $|N^{out}(\mathcal{A})| \geq |N^{in}(\mathcal{A})|$. Let t be a king in T with outdegree in T at least $\lfloor |T|/2 \rfloor$. Suppose that $|N^{in}(t) \cap N^{out}(\mathcal{A})| \geq |T|$. Then $\mathcal{A}$ is an SE winner.

Proof of Lemma 13: This proof will proceed by induction on the size of T. As such, we establish the base case when $|T| = 1$. Here, $T = \{t\}$ and $\mathcal{A}$ is actually a king in G with outdegree at least half the graph. By [143] $\mathcal{A}$ can win a single-elimination tournament.

Now consider when $|T| > 1$. Our induction proceeds by assuming that $\mathcal{A}$ can win if $|T| < p$ for some p, provided that $|N^{out}(\mathcal{A})| \geq |N^{in}(\mathcal{A})|$, t is a king of outdegree at least $|T|/2$ in T, $|T|$ is a power of 2 and $|N^{in}(t) \cap N^{out}(\mathcal{A})| \geq |T|$. Now given a graph with $|T| = p$, we will give a perfect matching M_G of the graph such that the following is true of the tournament G' induced by the sources of M_G:

1. if T_r are the surviving nodes of T, then $t \in T_r$ and t is a king in T_r of outdegree at least $|T_r|/2$ and $|T_r| = |T|/2$ is a power of 2,
2. if A_r are the surviving nodes of $N^{out}(\mathcal{A})$, then $in_{A_r}(t) \geq |T_r|$,
3. $\mathcal{A}$ is a king in $G' \setminus T_r$, and
4. if B_r are the surviving nodes of $N^{in}(\mathcal{A})$, then $|A_r| \geq |B_r|$.

In order to create the necessary matching M_G, first create a canonical matching $\mathcal{CM}(t)$ for t in T. Let T_r be the sources of $\mathcal{CM}(t)$. Then by Lemma 15, Condition 1 follows.

Now, let S be a subset of $N^{in}_{N^{out}(\mathcal{A})}(t)$ of size $|T|$. Create $\mathcal{M}(N^{out}(\mathcal{A}), N^{in}(\mathcal{A}) \setminus T)$. Since $|N^{out}(\mathcal{A}) \setminus S| \geq 1 + |N^{in}(\mathcal{A}) \setminus T|$, there exists an unmatched node a' in $N^{out}(\mathcal{A}) \setminus S$. We can match $\mathcal{A}$ to a'.

Next, match any unmatched nodes of S, $N^{out}(\mathcal{A}) \setminus (M' \cup S)$ or $N^{in}(\mathcal{A})$ amongst their respective sets. Call this matching M''. The number of nodes of S that survive is at least $\lfloor (|S| + |M'' \cap S|)/2 \rfloor \geq |S|/2 = |T_r|$. This satisfies Condition 2. Since M'' was maximal, all nodes of $N^{in}(\mathcal{A}) \setminus T_r$ have surviving inneighbors in A_r. This shows that $\mathcal{A}$ is a king in $G' \setminus T_r$, or Condition 3.

It remains to show that $|A_r| \geq |B_r|$. We know that

$$|A_r| \geq \lfloor (|N^{out}(\mathcal{A})| + |M''| - 1)/2 \rfloor$$

and that

$$|B_r| \leq \lceil (|N^{in}(\mathcal{A})| - |T| - |M''|)/2 \rceil + |T|/2 = \lfloor (|N^{in}(\mathcal{A})| + 1 - |M''|)/2 \rfloor.$$

Now, since $|N^{out}(\mathcal{A})| \geq |N^{in}(\mathcal{A})|$ by the induction hypothesis

$$|A_r| \geq \lfloor (|N^{in}(\mathcal{A})| + |M''| - 1)/2 \rfloor.$$

If $|M''| \geq 1$, we immediately get $|A_r| \geq |B_r|$. If $M'' = \emptyset$, then $N^{in}(\mathcal{A}) = T$. But then both $|N^{in}(\mathcal{A})|$ and $|N^{out}(\mathcal{A}) \setminus \{a'\}|$ are even. Furthermore, $|N^{out}(\mathcal{A}) \setminus \{a'\}| \geq |T| = |N^{in}(\mathcal{A})|$. Hence, $|A_r| = |N^{out}(\mathcal{A}) \setminus \{a'\}|/2 \geq |N^{in}(\mathcal{A})|/2 = |B_r|$. This proves Condition 4 and concludes the proof of the lemma. □

Reminder of Theorem 16 [**Not a King but Matching into $\mathcal{H}^{in}(\mathcal{A})$**] There exists a constant n_0 such that for all $n \geq n_0$ the following holds. Let $G = (V, E)$ be a tournament graph on n nodes, $\mathcal{A} \in V$. Suppose there is a matching M from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ onto $\mathcal{H}^{in}(\mathcal{A})$ of size K. If $K \leq (n-6)/7$, then $\mathcal{A}$ is an SE winner.

Proof of Theorem 16: This proof fleshes out the details that were ignored by the proof sketch given previously in the chapter but follows the same structure. It will be useful to refer to Figures 8.7 and 8.8 as we proceed through the construction.

For simplicity, let $A = N^{out}(\mathcal{A})$ and $B = N^{in}(\mathcal{A})$. We divide the sources of M onto $\mathcal{H}^{in}(\mathcal{A})$ into two sets, M_1 and M_2, where M_1 are the sources of M in A while M_2 are the sources in B. We can also divide $\mathcal{H}^{in}(\mathcal{A})$ into two sets, H_1 and H_2, where H_1 are the nodes matched to M_1 and H_2 are matched to M_2 by M. In order to later argue about the size of matchings, let $|M_1| = |H_1| = h$ and $|M_2| = |H_2| = k$. This means that K, the size of M is exactly $k + h$. Further, let $n_0 = 10^6$ although we suspect the theorem is true for much smaller n_0 with more careful analysis. Let C be the constant 529 and $c = \max\{C - k, 0\}$.

Let $\tilde{M}$ be an arbitrary matching of $B \setminus (M_2 \cup H)$ of size $\min\{c, \lfloor |B \setminus (M_2 \cup H)|/2 \rfloor\}$. We will call the set of sources of M_2 and $\tilde{M}$ B_T. Let $B_{\text{rest}} = B \setminus (B_T \cup H)$. Because B_T does not contain any nodes ranked higher than $\mathcal{A}$, for every $b \in B_T$ we have that $out_{B_T} + 2 \leq in_A(b)$. For our proof we will require that $|B_T| \geq C$. If $|B_T| < C$, then we will show how to use nodes from A to artificially boost the size of B_T, while still preserving the properties we need.

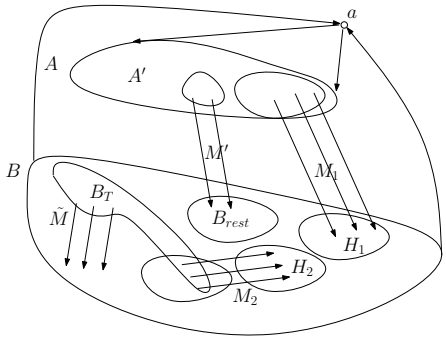

Figure 8.7: Situation in Theorem 16 when $Z = \emptyset$.

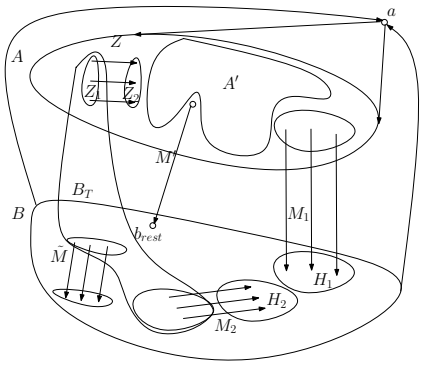

Figure 8.8: Situation in Theorem 16 when $Z \neq \emptyset$.

(Boosting $|B_T|$ when $|B_T| < C$.) If $|B_T| < C$, then $\tilde{M}$ was maximal and $|B_{\text{rest}}| \leq 1$. We now show that we can apply Lemma 18.

If $B_{\text{rest}} = \{b_{\text{rest}}\}$ and some inneighbor a_{rest} of b_{rest} is in $A \setminus M_1$, then note that

$$\begin{aligned} |A \setminus (M_1 \cup \{a_{\text{rest}}\})| &\geq \\ (n-1) - 2|M_1| - 2|B_T| - 2 &\geq \\ n - 3 - 2h - 2k - 2C &\geq \\ (5C^2 + 17C + 4)/2 & \end{aligned}$$

since $n \geq 10^6 \geq 7(10 + 21C + 5C^2)/10$ and $K < n/7$. Therefore,

$$\begin{aligned} n - 2h - 2k = n - 2K > 5n/7 &\geq \\ (5/7) \cdot 7(10 + 21C + 5C^2)/10 &= \\ (5C^2 + 17C + 4)/2 + 2C + 3 & \end{aligned}$$

This satisfies the conditions so that we can apply Lemma 18 to the sets A, $(A \cap M_1) \cup \{a_{\text{rest}}\}$, and B_T with the value C. This will give us a set $Z \subset A \setminus (M_1 \cup \{a_{\text{rest}}\})$ of size $2(C - |B_T|)$ in Z. Let $Z_1 \to Z_2$ be a (perfect) matching of size $C - |B_T|$ in Z. Add Z to B and Z_1 to B_T. Now we can assume $|B_T| \geq C$ and that for every subset $Q \subseteq B_T$ there are at least $\binom{|Q|}{2} + 2|Q|$ inedges of Q from $A \setminus Z$.

Let $\tilde{A} = A \setminus Z$, $\tilde{B} = (B \setminus H) \cup Z \cup M_2$ and $B_{\text{rest}} = B \setminus (\tilde{M} \cup M_2 \cup H)$. Since we are defining many sets, refer to Figures 8.7 and 8.8 for clarity. The figures cover the cases where $Z = \emptyset$ and $Z \neq \emptyset$ separately.

(Covering some of B_{rest}.) Let M' be a maximal matching from $\tilde{A} \setminus M_1$ to B_{rest}. There are several cases for this construction.

- If $Z \neq \emptyset$, M' is either empty, or only consists of $(a_{\text{rest}}, b_{\text{rest}})$. If $k \geq C$, then $|\tilde{M}| = 0$, $|B_{\text{rest}}| = |B| - k - K$. Furthermore, since every node b in $B \setminus (H \cup M_2)$ has at least

$out_B(b) + 1 - h = out_B(b) - (h-1)$ inneighbors from $A \setminus M_1$, by Theorem 17 we have that

$$\begin{aligned} |M'| &\geq (|B| - K - k - 2h + 2 - 1)/2 \\ &= (|B| - 2K - h + 1)/2. \end{aligned}$$

- If $Z = \emptyset$ and $k < C$, then $2(C-k)$ nodes of $B \setminus H \setminus M_2$ are matched to each other, and so by Theorem 17

$$\begin{aligned} |M'| &\geq (|B| - 2C + 2k - K - k - 2h + 2 - 1)/2 \\ &= (|B| - 2C - 3h + 1)/2. \end{aligned}$$

  Every node in B_{rest} has some source of M' or M_1 as an inneighbor.

(Matching $\mathcal{A}$ to some node.) Consider $A' = \tilde{A} \setminus (M_1 \cup M')$.

- If $Z \neq \emptyset$, then

$$\begin{aligned} |A'| &\geq n - 1 - |B| - |A \cap M_1| - 1 - |Z| \\ &\geq n - 2 - 2h - 2C \\ &\geq n - 2 - 2(n-6)/7 - 2C \\ &= (5n - 2 - 14C)/7 > 1 \end{aligned}$$

  Hence when $Z \neq \emptyset$, there is some $a' \in \tilde{A} \setminus (M_1 \cup M')$ that we can match $\mathcal{A}$ to.

- If $Z = \emptyset$. Then

$$\begin{aligned} |\tilde{A} \setminus M_1| &= \\ |A| - h &\geq \\ (n-K)/3 - h &\geq \\ (n - 4K)/3 &> 1 \end{aligned}$$

  If there is some $a' \in A'$, then match $\mathcal{A}$ to a'. Otherwise, $|M'| \geq 1$. Dislodge some edge (a', b') from M'. Since $out(\mathcal{A})$ and $in(\mathcal{A})$ have different parities and $A' = \emptyset$, the number of leftover unmatched elements of B after we add b' to them is even. Hence any matching we use to complete the first round of the tournament would be perfect on them. Even after removing a' from M' any surviving element b from the leftover B elements will have at least $out_B(b) \geq 1$ surviving inneighbors. The lower bounds we had computed for $|M'|$ go down by 1:

  - when $k < C$ and $Z = \emptyset$, $|M'| \geq (|B| - 2C - 3h - 1)/2$
  - when $k \geq C$, $|M'| \geq (|B| - 2K - h - 1)/2$ when $k \geq C$

Now let $S = A' \setminus \{a'\}$ and let T be the subset of B_T consisting of the nodes that do not have inneighbors among the sources of M' and M_1. Every subset Q of the nodes of T has at least $\binom{|Q|}{2} + 2|Q| - |Q| = \binom{|Q|}{2} + |Q|$ inneighbors in S since each node in Q can have lost at most one inneighbor, a'.

(Handling T' and completing round 1.) Let $t \in B_T$ be the node with highest outdegree in B_T. Running Algorithm 8 on S, T, t produces a matching M'' on the nodes of S so that almost all nodes of T are covered by sources S' of M'' except for a subset $T' \subset T$ with $|T'| \leq 1 + 2\sqrt{|T|}$. Further, there are at least $in_S(t)/2 - 1 \geq (|B_T| - 1)/4 - 1$ sources of M'' that beat t. The addition of M'' to the rest of our construction completes the first round matching. Call the graph induced by the surviving nodes G'.

Let P be the closest power of 2 greater than $3 + 4\sqrt{|T|}$. Then $P \leq 5 + 8\sqrt{|T|}$. Suppose that $|B_T| \geq 5 + 8\sqrt{|T|}$. This is true whenever $|B_T| \geq 81$, and since $|B_T| \geq C = 529 > 81$ the assumption is true. There exists a subtournament T_t of B_T such that $T' \cup \{t\} \in T_t$ and t is a king in T_t of outdegree at least $|T_t|/2$ and $|T_t| = P$, a power of 2.

If $t \in N^{out}(\mathcal{A})$, then we will not need its surviving inneighbors from S. In the following we handle the more complicated case when $t \in N^{in}(\mathcal{A})$, and so at least $in_S(t)/2 - 1$ inneighbors of t from S are in G'. We need that $(|B_T| - 1)/4 - 1 \geq 5 + 8\sqrt{|T|}$. This is true when $|B_T| \geq 529 = C$. Then there exists a subset of the surviving nodes of $N^{in}_S(t)$ of size at least $P = |T_t|$. Now we can apply Lemma 13 to show that $\mathcal{A}$ can win a single-elimination tournament, provided that $out_{G'}(\mathcal{A}) \geq in_{G'}(\mathcal{A})$.

(Showing $out_{G'}(\mathcal{A}) \geq in_{G'}(\mathcal{A})$.) Recall that $|M'| \geq (|B| - 2C - 3h - 1)/2$ when $k < C$ but $Z = \emptyset$ and $|M'| \geq (|B| - 2K - h - 1)/2$ when $k \geq C$. We have three cases.

1. $k \geq C$, and so $Z = \emptyset$.

   Here, the number of nodes of $N^{out}(a)$ that survive is at least $\lfloor (|A| + |M'| + |M_1| - 1)/2 \rfloor$. Meanwhile, the number of nodes of $N^{in}(\mathcal{A})$ that survive is at most $\lceil (|B| - |M_1| - |M'|)/2 \rceil$. It suffices to show that

   $$|A| + |M'| + |M_1| - 1 \geq |B| - |M_1| - |M'|.$$

   This happens when $|A| + 2|M'| + 2h - 1 \geq |B|$. By the assumptions of this case $|M'| \geq (|B| - 2K - h - 1)/2$, so we must show that

   $$|A| + |B| - 2K - h + 2h - 2 \geq |B|,$$

   or that

   $$|A| - 2K + h - 2 \geq 0.$$

   By Lemma 17, we know that $|A| \geq (n - K)/3$ so we just need that $(n - K) \geq 6K + 6$. This simplifies exactly to the assumption of the main theorem that $K \leq (n - 6)/7$.

2. $k < C$ and $Z = \emptyset$.

   In this situation, it still suffices to show that $|A| + 2|M'| + 2h - 1 \geq |B|$. However, now $|M'| \geq (|B| - 2C - 3h - 1)/2$. Combining these, we find we only need that $|A| - 2C - 3h + 2h - 2 \geq 0$, or equivalently that $n - 6C - 6 \geq K + 3h$. Simplifying this, we only need that $K \leq (n - 6C - 6)/4$, which is true since $(n - 6C - 6)/4 > (n - 6)/7$.

3. $Z \neq \emptyset$.

   If $Z \neq \emptyset$ then $|B| < h + 2C$, and at most C nodes of $B \cup Z$ survive. The number of nodes of $A \setminus Z$ that survive is at least

$$\begin{aligned} \lfloor (|A| - |Z| + h + |M'| - 1)/2 \rfloor &\geq \\ (|A| - C - 2 + K - C)/2 &= \\ (|A| + K)/2 - C - 1. & \end{aligned}$$

   We need only that $(|A| + K) - 2C - 2 \geq 2C$. After applying Lemma 17, this becomes that $(n - K)/3 + K \geq 4C + 2$, and $n + 2K \geq 12C + 6$. It is true that $n \geq 12C + 6$ since $n_0 > 12C + 6$.

This covers all of the cases and concludes the proof. We have given the construction for a matching $M \cup \tilde{M} \cup M' \cup M''$ such that the conditions for Lemma 13 apply to the node $\mathcal{A}$ in the subtournament induced over the sources of our matching. □

We can state this result in terms of the size of $\mathcal{H}(\mathcal{A})$ instead of $\mathcal{H}^{in}(\mathcal{A})$ by applying Lemma 16 to lower bound the size of the initial set A.

Corollary 4. *There exists a constant n_0 so that for all tournaments G on $n > n_0$ nodes the following holds. Let $\mathcal{A}$ be among the top $(6n+7)/31 \geq .19n$ highest outdegree nodes. If there is a matching from $V \setminus \mathcal{H}^{in}(\mathcal{A})$ onto $\mathcal{H}^{in}(\mathcal{A})$, then $\mathcal{A}$ is an SE winner.*

Chapter 9

Double-Elimination Tournaments

9.1 Introduction

Voting mechanisms have been studied mathematically for over two hundred years. A combination of impossibility results, including Condorcet's paradox, Arrow's Impossibility Theorem [19] and the Gibbard-Satterthwaite Theorem [61, 128] have shown that there is no one best voting mechanism for all situations and goals. This has led to the creation of large number of election protocols, including k-approval, Borda, Buckland, Copeland, Slater, instant run-off, single-tranferable vote, binary cup, and many many more. The recent interaction between economics and computer science has led to more computationally focused questions. Some examples are the following. Can the winner of the protocol be calculated efficiently? Some protocols require solving NP-complete problems, so efficiency there is unlikely. Can a manipulation of the protocol be found efficiently? The Gibbard-Satterthwaite theorem states that all non-trivial voting mechanisms are manipulable, but if it is hard to find the manipulation, then these mechanisms are perhaps not manipulable in practice.

Any tournament structure can be used as an election protocol if the match outcomes between each pair of candidates are generated by taking the majority vote. Sports competitions are a special kind of voting mechanism where the votes are generated exogenously, perhaps by skill differentials between the candidates. In the United States alone, professional and amateur sports competitions form a multi-billion dollar industry. Their popularity and ubiquity makes the study of their properties particularly interesting.

While all voting mechanisms share a set of common goals - they should be 'fair' and select the best preferred or strongest candidate, sports tournaments have a few additional constraints. In particular, each 'vote' actually consists of a match. These matches require some amount of time and have limited parallelizability due to either space limits or dependence on the results of previous rounds. Another constraint is that these match outcomes are noisy. One traditional solution is to turn a single match into a 'best-of-k'-series, as in the Major League Baseball World Series. However, repeating games to reduce noise directly conflicts with the time constraint. Finally, we require that the mechanisms be equally fair

to all players.

Clearly, the time constraint and robustness constraint cannot be simultaneously satisfied, leading to a trade-off between the two. At one end of the spectrum is the *round-robin* tournament where every player faces every other player. Braverman and Mossel [32] have shown that a good ranking can be recovered from the results, but at the expense of spending time that is quadratic in the number of players. At the other end of the spectrum, *single-elimination* tournaments (SETs) eliminate any player after a single loss, requiring only a linear number of games, but are highly susceptible to noisy outcomes. Stanton and Vassilevska Williams [137] have shown how to exploit this noise so that almost any player can be made to win an SET.

Double-elimination tournaments (DETs) occupy a natural middle ground between these two options. DETs consist of two SETs, the winner bracket and loser bracket, where players who lose in the winner bracket are mapped to the loser bracket. When players lose in the loser bracket, they are permanently eliminated. The winner of the tournament is the player who wins the matchup between the winners of the two brackets. DETs still require only a linear number of matches, but the second chance given to each player naturally reduces the noise in the outcome.

DETs are widely used in the real world, albeit unevenly. Their most prominent use in the United States is as the format for many baseball and softball tournaments, including by the National Collegiate Athletic Association (NCAA), as well as in numerous amateur competitions and Olympic sports such as Judo. It is a particularly popular format for children's competitions because it allows each team to play more games than in a SET. Besides their use in sports events, DETs are often used for important applications from decision making in multi-agent systems to medical trials. For over forty years, DETs have been used to fit hearing aids [10]. They are also used in experiment design [122] and are featured in many patents [92, 48]. Given their ubitquity, the study of the structure and properties of DETs is an important question. To our surprise, we found very little attention has been paid to DETs by mathematicians, staticians, economists or computer scientists. In particular, we found that there is not even a standard definition for a DET.

9.1.1 Contributions

We investigate the structure of DETs with respect to several design goals: *balance, fairness*, and *repeat-avoidance*. Balance and fairness are addressed in the same way in all DET designs used in practice, via the tournament structure. Avoiding repeated matches is a more complex problem. It is addressed by most DET designs in different ways. We develop an effective way of analyzing different structures for their repeat-avoidance properties.

We study the link functions used in practice and show that they are all formed with the same two basic primitives. We show that the link functions based on these primitives used in practice are not optimal with respect to avoiding repeated matches and propose a related link function that is optimal. As the full structure of DETs has not been universally defined, we believe that this work formalizing the definition of the structure of DETs is a crucial first

step before we can study computational questions related to DETs.

We initiate the study of the complexity of manipulating DETs for a given structure (winner and loser bracket structure and a link function). We show that DETs are vulnerable to manipulation by players who can improve their chance of winning by throwing a match. We show that coalition manipulation of DETs can be computed in polynomial time for some link functions. A corollary to this proof gives the first polynomial time algorithm for calculating the probability of a given player winning a DET for a very specific link function. We also discuss manipulation by a tournament organizer (agenda control) in two settings. The first is by changing the player seeding in the winner bracket. This is the same type of manipulation that can be used against SETs. The second type of manipulation allows the tournament organizer to pick the link function. We show that with respect to seeding manipulation, SETs and repeat-avoiding DETs are similar. Additionally, if DETs are not required to be balanced, then manipulating the outcome by picking the link function is NP-hard. For each type of manipulation, we formulate interesting open questions.

In the Appendix, we empirically investigate the effect of the link functions on the outcome of tournaments generated by two natural stochastic models. We determine that using a repeat-avoiding link function does not negatively affect whether the tournament selects a good winner, justifying our DET design. We also demonstrate that DETs are much better at selecting strong players than SETs in our models.

9.1.2 Related Work

SETs are well-studied by both mathematicians and statisticians [17, 39, 36, 73, 100, 101], and in the area of computational social choice [72, 86, 69, 144, 143, 138, 26]. DETs have been studied largely by statisticians focusing on their structure [52, 100, 129, 85]. There are also a number of patents on using DETs for a range of applications from sports competition design to gambling, to design of drug trials.

With regards to our experimental section in the appendix, [100, 64, 60, 123] concern themselves with the experimental evaluation of tournaments. With the exception of [60], these papers are restricted to round-robin and SETs, and evaluate the effectiveness of either prespecified or random seedings and their effect on the tournament outcome, usually for the linear stochastic model of match outcomes for tournaments with either 4, 8 or 16 participants. Our experiments in the appendix focus on random seedings for DETs with multiple stochastic models and between 8 and 1024 players.

The question of design of knockout tournaments has previously been considered [5]. However, they consider designing new types of competitions which, unfortunately, are not used in practice. Instead, we focus on a tournament type that is popularly used and investigate its efficacy at its stated goal.

9.2 Formal Definition of Double-Elimination Tournaments

The structure of a DET has been previously defined by [52] as *any* (not necessarily balanced) SET on N players, the losers of which play in a *loser* bracket which is *any* SET on $N-1$ players. The winners of the two brackets then play each other in the final to determine first and second place. Besides the two brackets, there is a mapping, called the *link function*, which defines where exactly in the loser bracket the losers of the winner bracket are placed. In the definition of a DETin [52], neither the bracket structure, nor the link function are fixed, as the focus of the paper is to count the number of different DETs of a given size.

In this chapter we fix the structure of both the winner and the loser brackets, and we study several link functions. In order to understand the choice for the DET structure we first discuss the main design goals.

balance There should not exist seeding positions where it is inherently easier for the seeded player to win. An interpretation of this goal is that the tournament structure should be such that no matter where a player is seeded, the number of matches that the player needs to win in order to win the tournament should be the same. For instance, a SET on N players with a single match in each round does not obey this goal – the players who play the first match must win $N-1$ matches to emerge victorious, while the last player must only win 1.

This design goal is easy to achieve for a SET: the bracket structure should be a balanced binary tree. For DETs this goal is not easy to achieve because the number of opponents a player faces depends on the round where they lose in the winner bracket. One can start by making the winner bracket balanced, as are all DET designs in practice. However, picking the structure of the loser bracket and the link function is more complex. We instead focus on a weaker requirement, round-fairness.

round-fairness Let us assume that the winner bracket is balanced. This is a common practice as tournaments can be made balanced after the first round through the use of *byes*, which grant an automatic advance to the next round for the selected players. Consider two players who lost in the same round of the winner bracket and have an equal number of wins in the loser bracket. We would like these players to compete against opponents of similar quality in their next match. This implies a constraint not only on the loser bracket structure but also on the link function. We will define a new balance structure for the loser bracket which is uniformly used in practice. The link function then needs to map players who lose in the same round of the winner bracket to the same round of the loser bracket.

Suppose for simplicity that the number of players N is a power of 2. The tournament structure that we will use is that the winner bracket is a balanced SET with $n = \log(N)$ rounds, while the loser bracket has $2n$ rounds. The tournament starts with Round 1 of the winner bracket. Round 1 of the loser bracket is played after round 1 of the winner bracket. During odd rounds of the loser bracket, players in the loser bracket are matched against

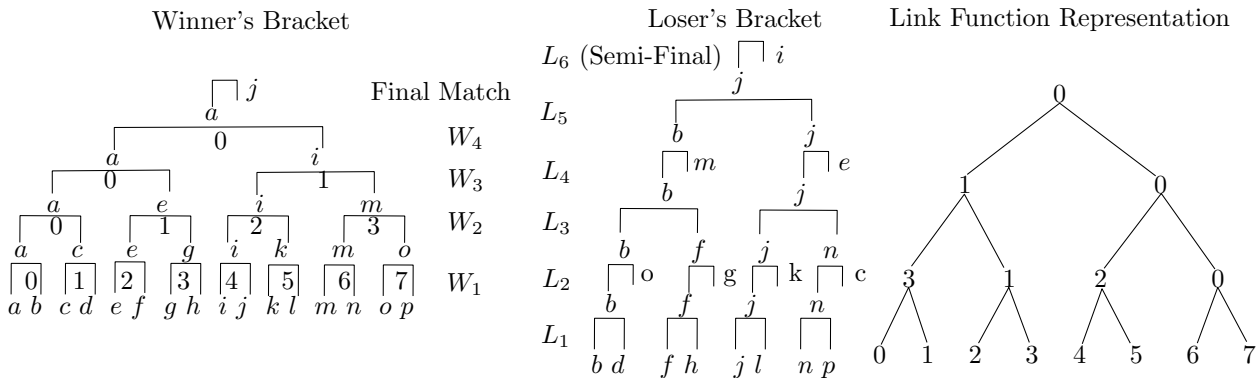

Figure 9.1: DET structure for 16 players: The left is a balanced SET giving the winner bracket, while the right is the loser bracket in which the round i winner bracket losers are seeded at round $2i-2$. The loser bracket rounds are labeled L_i while the winner bracket rounds are labeled W_i. The rightmost tree represents the link function in this example using the notation $\mathcal{T}_f$ discussed in Section 2.1. The labels are not given in binary for space reasons.

each other, while even rounds $2(i-1)$ have the losers from Round i of the winner bracket matched according to the link function against the loser bracket players who survived round $2(i-1)-1$. Notice that in round i of the winner bracket, $N/2^i$ players are eliminated and are mapped to round $2i-2$ of the loser bracket. Odd rounds $2(i-1)-1$ of the loser bracket contain $N/2^{i-1}$ players, $N/2^i$ of which win their matches and make it to loser bracket round $2i-2$ where they face the $N/2^i$ losers from the winner bracket round i. The final match of the tournament is between the two bracket winners. If the winner bracket loser loses in the final, a rematch is played until there is only one player left with at most one loss. This construction satisfies round-fairness and is used in practice. See Figure 9.1 for an example of this structure. Further examples of the structure can be found at [118].

repeat-avoidance In SETs, no match is ever repeated since losers are always eliminated from the tournament. Other tournament structures, such as DETs and Swiss tournaments can contain repeated match-ups. In practice, such repeats are explicitly avoided. One way to avoid them is for the tournament organizer to reseed teams at each round 'on-the-fly' if a repeat is about to occur. This however gives the tournament organizer an immense amount of power to change the tournament outcome. This is especially true in later rounds of the tournament when there are fewer players and brute-force approaches become easier. Because of this, one would prefer for the tournament structure itself to minimize the possibility of repeats.

In practice, tournament structures and seedings are explicitly designed so that repeats are minimized. For example, existing Swiss tournament design software such as SQBS and tsh [131, 37] devote a significant fraction of their configuration details to avoiding repeated matches. In tsh [37], one can specify the number of rounds that must pass before any match

is repeated. The documentation also notes that repeats become unavoidable at later rounds of the tournament. Many other sports organizations explicitly mention minimizing repeats in their tournament rules [116, 84, 142]. Similarly, the strict definition of Swiss Tournaments in [17] disallows any repeats of games. [17] also discusses DETs under the name 'draw-and-process', and explicitly forbids repeats until the semi-final match.

There are many reasons to avoid repeated matches. Repeats could be beneficial if *all* matches are repeated the same number of times, and the repeats confirm that one team is better than the other. If only certain games are repeated, the tournament is unfair to the losers of the unrepeated matches, since only some losers have a chance to redeem themselves. Also, if a match is repeated and the outcome changes, it is not always possible to conclude much more than 'these teams are approximately even'. Because of this, repeated matches can be quite controversial. A recent example is the 2011-2012 BCS National Championship Game of Alabama versus Louisiana State (LSU). LSU entered with an undefeated record having previously faced Alabama and won. In the championship game Alabama won, leaving each with a tied record of a single loss to the other team. Alabama was declared the national champion, despite the tied record.

Repeats early on in the tournament are most often between unequal teams and such repeated matches are boring for both the participants and the spectators, as it is not very interesting to see one team dominate another repeatedly. If repeats are to occur, one would want them to be in the later stages of the tournament when strong teams play each other. This motivates the 'draw-and-process' discussion in [17] and, as we will see later, this also motivates our definition of repeat-avoidance. A final reason to avoid repeats is due to time constraints: DETs only have $2N-1$ matches, and it makes sense to have as many diverse match-ups as possible in order to learn as much as possible about the underlying total order of the players.

The goal of round-fairness can be easily satisfied by restricting the link function and using symmetric and balanced structures for both the winner and loser brackets. The goal of balance is approximately satisfied since any tournament winner must win between $n+1$ and $2n$ matches. Repeat-avoidance, on the other hand, entirely depends on the link function.

9.2.1 Link Function Notation

To define a link function, we need a standard way of referring to players and their paths through the winner bracket. We label the players in round ℓ of the winner bracket from left to right by 0 to $2^{n-\ell+1}-1$ in binary. Note that in round ℓ of the winner bracket, any pair of players who play each other have identical prefixes of length $n-\ell$, e.g. in round 1, 0^n plays $0^{n-1}1$. The player who wins moves up in the winner bracket and adopts the prefix of length $n-\ell$, while the player who loses moves to round $2\ell-2$ of the loser bracket, also adopting the prefix as its label.

There are exponentially many valid link functions. With $N=2^n$ players, $2^{n-\ell}$ players lose in round ℓ of the winner bracket. A link function f_n is valid if its restriction to round $(2\ell-2)$ of the loser bracket, $f_n(\ell)$, is any permutation of the numbers $0,\cdots 2^{n-\ell}-1$.

There is a natural representation of any link function f into the loser bracket as a labeled balanced binary tree $\mathcal{T}_f$. The nodes at level ℓ of $\mathcal{T}_f$ represent the losers of the round ℓ winner bracket matches and are labeled from left to right by $f(\ell)$. In this representation, each subtree is a subtournament of the loser bracket. Any node of $\mathcal{T}_f$ plays the winner of the match between the winners of the subtournaments rooted at its children. See Figure 9.1 for an example of $\mathcal{T}_f$.

9.2.2 Repeat-Avoidance

Repeat-avoidance is an intuitively clear goal with an unfortunately large number of potential mathematical definitions. We will assess the repeat-avoidance quality of link functions in terms of the first round of the loser bracket in which a potential repeat may occur, or equivalently, according to the number of players that remain in the loser tournament when the first potential repeat can occur. Our definition was motivated earlier in the introduction. One can show that one cannot avoid potential repeats, and so we aim to minimize them. We focus on potential repeats since we do not want to assume anything about the match outcomes. Additionally, our definition of repeat-avoidance is used implicitly in practice in the design of tournaments by both [142, 37]. In the tsh documentation, our definition is referred to as 'monagony' [37].

One of the benefits of viewing the link function as a labeled binary tree is that it is easy to detect repeat matches. In a subtree of height ℓ of the tree $\mathcal{T}_f$, two players *may* have previously played if they have the same $n - \ell$ bit prefix. Therefore, without assuming anything about the underlying match outcomes, we can analyze a link function in terms of prefix collisions at various tree levels. The later the level where collisions occur, the better the link function is in terms of repeat-avoidance.

Swaps and reverses. While the link function is not standardized, most real-world tournaments (see [118]) use the same two primitives for the construction of the link function, the functions *swap* and *reverse* as given in Definition 5. These definitions are given as a left-to-right labeling of the nodes in each even level of the loser bracket using the prefix labels of the winner bracket.

Definition 5. ***[Link function primitives]*** *Define two functions s and r which each take a list of k numbers $a_1, \ldots, a_k$ where k is a power of 2 as:*

$$s(a_1, \ldots, a_k) = a_{\frac{k}{2}+1}, \ldots, a_k, a_1, \ldots, a_{\frac{k}{2}}$$
$$r(a_1, \ldots, a_k) = a_{\frac{k}{2}}, \ldots, a_1, a_k, \ldots, a_{\frac{k}{2}+1}$$

s splits the list in half and swaps the pieces, while r splits the list in half and reverses each part in place.

We can also define a generic function that uses these primitives recursively. The following function takes as input a string of instructions, consisting of r's and s's, applies the first one,

and then recurses on each half of the string after stripping the first character off of its instruction string.

Definition 6. ***[Generic link function definition]*** *Given a string l in the regular language $\{(r,s)^k | 0 \leq k \leq \log n\}$, define $\hat{f}$ as*

$$\hat{f}(a_1 \dots a_n, l) = \begin{cases} \hat{f}(a_{\frac{n}{2}}, \dots, a_1, l[1:L(l)]) \odot \hat{f}(a_n, \dots, a_{\frac{n}{2}+1}, l[1:L(l)]), \text{if } l[0] = r \\ \hat{f}(a_{\frac{n}{2}+1}, \dots, a_n, l[1:L(l)]) \odot \hat{f}(a_1, \dots, a_{\frac{n}{2}}, l[1:L(l)]), \text{if } l[0] = s \end{cases}$$

where $l[1:L(l)]$ is the string l with its first character removed, $L(l)$ is the length of the string and $\odot$ is string concatenation. If l is the empty string, $\hat{f}$ returns the input unchanged.

Any function that uses s or r exclusively is not ideal i.e. one where swaps are used every round and never reverses or vice versa. We can see that it avoids repeats until round $n/2$ of the loser bracket (or until $\sqrt{N}$ players remain) by the following observation. In each consecutive level of $\mathcal{T}_f$ each node differs from all of its descendants in at least one bit. Meanwhile, the number of bits in the bit representation is reduced by one. Repeat-avoidance can proceed up to level ℓ, as long as the number of bits in the representation, $n - \ell$, is at least the level number ℓ, and hence only while $\ell \leq n/2$. One can see that this argument is tight.

By contrast, the link functions used in practice alternate using exclusively swaps or reverses for each round. In Theorem 18 below, we show that the most common way to alternate the two primitives, improves the repeat-avoidance property from $\sqrt{N}$ remaining players to $N^{1/3}$ remaining players.

The most commonly used link function is below and other functions used in practice change the ordering of which rounds use the swaps or the reverses. The pattern below is the one used for instance by [118].

Definition 7. ***[Link function in Practice]*** *Call this function F. For every $k \leq 2(n-1)/3$, $F(k)$ produces a list that labels the kth level of $\mathcal{T}_F$ where:*

$$\begin{aligned} F(1) &= 1^{n-1}, \dots, 0^{n-1} \\ F(2k) &= \hat{f}(0^{n-2k}, \dots, 1^{n-2k}, s^{\min\{k, n-2k\}}) \\ F(2k+1) &= \hat{f}(0^{n-2k-1}, \dots, 1^{n-2k-1}, r^{\min\{k, n-2k-1\}}) \end{aligned}$$

Theorem 18. *F avoids repeat games until at most $N^{1/3}$ players remain.*

Proof. Let ℓ be the largest interger such that $\lfloor \ell/2 \rfloor \leq n - \ell$. Consider level $i \leq \ell$ of $\mathcal{T}_F$. By definition, $F(i)$ is an ordered list of $(n-i)$-bit binary strings. Because the only operations used are swaps and reverses, the list is partitioned into consecutive pieces on $2^{n-i-\lfloor i/2 \rfloor}$ binary strings which share the same prefix of length $\lfloor i/2 \rfloor$. This $\lfloor i/2 \rfloor$-long prefix of a node at level $i \leq \ell$ of $\mathcal{T}_F$ differs from the prefixes of all of its descendants: All even levels of $\mathcal{T}_F$ were formed by the function r and all odd levels by s, and thus the first bits of the even and odd

levels differ; between two even or two odd levels, some other bit must differ because an extra swap or reverse stage was performed. Therefore, every node in a subtree rooted at level ℓ has a different prefix of length $L = \lfloor \ell/2 \rfloor$ from its descendants. This is only possible while $L \leq n - \ell$, i.e. $\ell \leq 2n/3$.

To see that at level $\ell + 2$ there will be repeated prefixes, note that in order to get a difference in the prefixes between a level i and level $i - 2$, one needs to do at least one more swap or reverse at level i compared to level $i - 2$. This is why the number of swap/reverses is roughly $i/2$ and no less. At level $\ell + 2$ the number of strings available is $2^{n-2-\ell}$ and hence the number of possible swaps or reverses is at most $n - 2 - \ell \leq L$, while at level ℓ there were L swaps or reverses. Hence each node at level $\ell + 2$ has the same prefix as one of its descendants. □

We give a lower bound for repeat-avoidance:

Claim 1. *For any link function f, a repeat will occur at round $n - \log n + 1$ of $\mathcal{T}_f$, or when $n = \log N$ players remain.*

Proof. Notice that each level i of $\mathcal{T}_f$ that avoids a repeat must contribute at least one unique prefix of length $n - i$ to the list of prefixes to avoid in round i. Since there are at most 2^{n-i} distinct prefixes of length $n - i$, if $i > 2^{n-i}$, then there will be repeats. This happens for $i > n - \log i$, i.e. definitely at round $n - \log n + 1$. □

Given the above limitation of link functions, we show how to construct optimal ones.

Theorem 19 (Optimal Link Functions)**.** *One can construct many link functions that avoid any repeats until* $\log N$ *players remain.*

Proof. Let f be a link function. Consider all levels i of $\mathcal{T}_f$ for which $i \leq 2^{n-i}$. Let ℓ be the last such level. Then $\ell \geq n - \log n$.

In order to avoid repeats up to level ℓ, it suffices that no subtree of $\mathcal{T}_f$ rooted at a node at level ℓ contains repeated prefixes on $n - \ell$ bits. Consider an ℓ by $2^{n-\ell}$ matrix such that each row contains all the numbers from 0 to $2^{n-\ell} - 1$, and no column contains a repeated number. Many matrices of this form exist. For instance, take any permutation π of the numbers from 0 to $2^{n-\ell} - 1$. A matrix can then be formed by taking π and $\ell - 1$ of its rotations by one element as the rows of A. This is always possible as long as $\ell \leq 2^{n-\ell}$.

Each such matrix A can be converted into a link function f for which there are no repeats before level $\ell + 1$ as follows. For each $i \leq \ell$, let $f(i) = A[i, 1]\{0, 1\}^{\ell-i}, \ldots, A[i, 2^{n-\ell} - 1]\{0, 1\}^{\ell-i}$, where $A[i, j]\{0, 1\}^{\ell-i}$ is the list

$$A[i, j] \odot 0^{\ell-i}, A[i, j] \odot 0^{\ell-i-1}1, \ldots, A[i, j] \odot 1^{\ell-i}$$

with $A[i, j]$ written in binary. One can pick any setting for $f(j)$ when $j > \ell$. Notice that the descendants of a node x in level $i < \ell$ in $\mathcal{T}_f$ have completely different prefixes from x. Because of this, there can be no repeats till level ℓ. At this point there are at most $2 \log n = 2 \log \log N$ rounds of the loser bracket left. □

This bound is asymptotically tight and any labeling that consists of permutations of prefixes of length $n - 1 - \log n$ so that no column contains a repeated number, is optimal with respect to repeat-avoidance.

We can easily create a link function similar to those used in practice that is optimal with respect to repeat-avoidance by using more distinct patterns and interleaving the swap and reverse functions. For example, we can now see that the primary issue with 'swaps-only-or-reverses-only' patterns previously shown to only avoid repeats to $O(\sqrt{N})$ rounds is that there are only $2n = 2\log N$ distinct possible patterns of length n in the language $\{r^k | 0 < k \leq n\} \cup \{s^k | 0 < k \leq n\}$. If, instead, we use all patterns contained in the regular language $\{s, r, \epsilon\}^n$, then we have 2^n different patterns for use in defining the link function rounds. Using this, we now define an optimal link function.

Definition 8. ***An Optimal Swap-Reverse Link Function.*** *Let the language of strings $P = \{s, r, \epsilon\}^{\log n}$ be lexicographically sorted such that $P[i]$ refers to the i^{th} string. Now, the link function, $\hat{g}$, for $N = 2^n$ players can be defined as:*

$$\hat{g}(i) = \begin{cases} \hat{f}(0^{n-i}, \ldots 1^{n-i}, P[i]) & i < n - \log n \\ \hat{f}(0^{n-i}, \ldots 1^{n-i}, P[n-i]) & i \geq n - \log n \end{cases}$$

In order to establish that this function $\hat{g}$ is optimal, we need to show that enough distinct patterns are used, and that none of these patterns generate the same string. The first observation is easy - the set P has size $2^{n-\log n}$. The second observation relies on noting that each operation works on a recursively smaller portion of the sequence. For example, if swap is the first operation applied, then, regardless of the later operations, only $10^{n-1} \ldots 1^n$ can appear in the left half of the output, while reverse fixes this half of the output to consist of $0^n, 01^{n-1}$. Therefore, for any pair of strings $s_1, s_2 \in P$, let i be the first position where they differ. If we look at the first $n/2^i$ entries of the input string at the i^{th} operation, s_1 and s_2 will, by the argument above, force the two halves of this portion of the string to differ in every entry. Applying further operations on each half independently cannot cause these to collide again. Therefore, their outputs must differ in every position and there are no repeats.

The final link function we will mention is the identity link function. When the match outcomes are deterministic, the identity link function causes a DET to behave exactly like an SET.

Definition 9. ***Identity Link Function*** *Let I denote the identity link function. For every $1 \leq i \leq n$,*

$$I(i) = 0^{n-i}, 0^{n-i-1}1, \ldots 1^{n-i}$$

The identity link function has immediate repeats beginning in round 2 of the loser bracket.

9.3 Manipulation in Double-Elimination Tournaments

The manipulation of voting protocols is a major focus of the area of computational social choice. The fact that any non-trivial voting protocol is manipulable is guaranteed by the Gibbard-Satterthwaite Theorem [61, 128], although Bartholdi, Tovey and Trick [24] introduced the idea of using computational complexity to categorize how manipulable each protocol is.

In this section, we initiate the study of the manipulability of DETs and define several open problems for further study. We consider two types of manipulation, by the players themselves and by the tournament organizer (agenda control). Both types of manipulation occur in real tournaments. Most notably, in the 2012 Olympics, four of the top badminton teams were disqualified for trying to intentionally lose matches, causing an uproar and angering fans. While the tournament structure used there was not a DET, this example demonstrates that players really will exploit poor tournament design when possible.

In both types of manipulation, we study the computational problem of determining whether a manipulation is possible. The input to all versions of the problem includes complete deterministic information about the outcomes of all possible match-ups between pairs of players, i.e. for every pair u, v one is given whether u would beat v or v would beat u if they play against each other. A probabilistic version of this information is discussed in one of our cases. The deterministic match outcome information can be represented as a tournament graph in which the nodes are the players and a directed edge (u, v) means that u would beat v.

9.3.1 Manipulation by Players

In SETs it is never in a player's best interest to intentionally lose a match. In DETs a player is not eliminated until they lose a second time, so it may be advantageous to lose a match if the link function maps the player to a more favorable place in the loser bracket. We give a specific example of this behavior using the link function F defined earlier: Suppose that the winner bracket seeding is (from right to left), $abcdefghijklmnop$ so that a, c, f, h, j, l, n, p make it to the second round of the winner bracket, while the initial seeding of the loser bracket is $bdegikmo$. Suppose that m beats $\{o, a, c\}$, d beats a, a beats c, c beats everyone else, and everyone else beats m. If c beats d, it will be eliminated by a and m. However, if c throws the match with d then a will face d and lose, and then face m and be eliminated. In the next round m will be eliminated and c beats all of the remaining players and wins the tournament.

A more general case of player manipulation is when there is a coalition C of players that, given a seeding and full knowledge of the match outcomes, can make a decision about which matches to lose intentionally with the goal of making a particular player the tournament winner. The complexity of deciding whether such a manipulation is possible for a given seeding and a tournament graph describing all pairwise match outcomes has been studied in the context of both SETs and DETs by [124]. They showed that for SETs the problem can be

solved in polynomial time, regardless of the size of the coalition C. Their result for DETs is weaker. They showed that if $|C|$ is a constant, then the manipulation can be computed in polynomial time. We give an improvement on the Russell and Walsh argument by showing that if $|C| = O(\log n / \log\log n)$, then one can find an optimal manipulation strategy for a DET with n players in polynomial time.

Theorem 20. *There is a $2^{O(c \log\log n)}$ time algorithm which solves the coalition manipulation problem for a DET on n players and coalition of size c.*

Proof. Consider any player v in the coalition C. There are at most $2\log n$ matches that v can play in a balanced DET: up to $\log n$ in the winner bracket and up to $2\log n$ in the loser bracket. Since v can only lose twice before v is eliminated, there are at most $2\log^2 n$ choices of the matches in which v can lose. Now consider the players in C in some order, $p_1, \ldots, p_c$. For each i, and every choice of ways to pick matches to intentionally lose for the first $i-1$ players, consider how the tournament would play out if those matches were actually lost intentionally. There are still at most $2\log^2 n$ choices of matches that player p_i can intentionally lose. Hence, for c players, there are at most $O((2\log^2 n)^c) \leq 2^{O(c \log\log n)}$ manipulation choices, and each choice can be checked in polynomial time. □

This result and the one in [124] do not depend on the link function since the argument is that for small C there are only a polynomial number of ways the coalition can manipulate the tournament outcome. It is conceivable that for larger coalitions finding the best manipulation greatly depends on the link function. We show that for the simplest link function, the identity link function, coalition manipulation is in polynomial time, regardless of $|C|$. The argument breaks down for more complicated link functions and we conjecture that for some link function the problem is hard.

Theorem 21. *Let C be a coalition of players in a DET with the identity link function. Computing an optimal manipulation strategy for C to pick the winner of the tournament can be done in polynomial time.*

The proof appears in the Appendix. It proceeds by providing an algorithm for manipulation using an intricate dynamic programming approach. The proof technique also implies the following two corollaries.

Corollary 2. *The set of possible winners of a DET with the identity link function can be computed in polynomial time.*

Corollary 3. *Given all pairwise probabilities of the outcomes of each match where p_{xy} is the probability of player x beating player y and $p_{xy} + p_{yx} = 1$, the probability of a player winning a DET with the identity link function can be computed in polynomial time.*

The argument breaks down for more complicated link functions because the independence between the subtrees of the loser brackets is lost. We leave two open problems.

Open Problem 1. *Is the coalition manipulation problem for DETs in polynomial time for all link functions?*

Open Problem 2. *Can one compute the winning probability of a player in polynomial time for any link function?*

We conjecture that for some link function either computing the winning probabilities, or computing coalition manipulation is hard.

9.4 Efficacy of DETs at Selecting Strong Players

Recent results in SET manipulation have shown that tournaments can be manipulated for very weak players with high probability in random tournament generating models [143, 137]. These results lead us to question the quality of winner selected by SETs and, in turn, wonder if DETs suffer from similar weaknesses. We will investigate this question from both a theoretical and empirical viewpoint and show that, empirically, DETs are much more robust than SETs when faced with noisy comparisons.

In order to investigate this problem, we must establish a ranking. Given a tournament graph which summarizes the match outcomes, we use the ranking generated by sorting the players by their number of wins, with ties handled arbitrarily. This ranking is a 5-approximation of the one generated by solving Feedback Arc Set [44] which, in turn, is equivalent to Slater Voting. When using this ranking, it is known that the strongest player can always win an SET, and that the top 19% can if there exists a matching onto the stronger players [138].

First, consider how weak a player can be and still win a DET. It is necessary that a player beat at least $1 + \log N$ players because the winner bracket consists of $\log N$ rounds, and then the player must also face the winner of the loser bracket. It is easy to construct a seeding and match outcomes where the player who wins exactly meets this bound. The ranking of this player could be as low as $N - 2\log N$.

Before introducing the next example, we define the identity link function. When the match outcomes are deterministic, the identity link function causes a DET to behave exactly like an SET. The identity link function has immediate repeats beginning in round 2 of the loser bracket.

Definition 10. ***Identity Link Function*** *Let I denote the identity link function. For every $1 \leq i \leq n$,*

$$I(i) = 0^{n-i}, 0^{n-i-1}1, \ldots 1^{n-i}$$

Now, consider a player at the top of the ranking. As a simple example, let player a beat everyone except b who beats everyone except c. Let c be beaten by everyone except b. a and b have identical outdegrees, and both can win SETs. However, a can only win a DET that uses the identity link function. This is because a can only win if b is eliminated, and only c can eliminate b. If the link function used has any amount of repeat-avoidance, after c beats

b in the first round and then loses in the second, it is seeded far away from b. This shows that the strongest player may not be able to win a DET.

These simple examples demonstrate that DETs may perform quite poorly. However, these are very specific counterexamples and are unlikely to occur in practice. As such, we are also interested in the average case. This question has previously been studied for DETs, but in the context where the ranking of the players is known in advance. [60] show that DETs have very good outcomes when the ranking is known. However, we believe that this is an extremely strong assumption. If the ranking is already known, then do we hope to learn by running a tournament? Our study is motivated by [100] who studies the probability of the top player winning an SET when given a random seeding versus the standard seeding. The standard' seeding is where in the first round, 1 plays N, 2 plays $N-1$ etc.

9.4.1 Experimental Results

Size	CR-Log	CR-Sqrt	Linear	Size	CR-Log	CR-Sqrt	Linear
8	0.00548	0.0039	0.00438	128	0.00115	0.00101	0.0042
16	0.00627	0.00502	0.00469	256	0.000668	0.00067	0.00546
32	0.00318	0.0033	0.00564	512	0.000377	0.000515	0.00398
64	0.00192	0.00229	0.00466	1024	0.000325	0.000730	0.00721

Table 9.1: Average total variation distance for the two link functions for each model

In this section, we answer two questions. The first is whether making the tournament more fair by changing the link function from the one used in practice to the optimal link function reduces the quality of the outcomes of the tournament. The second is why, if DETs take twice as long as SETs, and are similarly susceptible to manipulations, one would choose to use a DET. In short, the answer to the first question is that the empirical distributions generated by changing the link functions are practically indistinguishable, and the answer to the second is that DETs significantly boost the probability that the first player will place first and the second will place second over SETs.

The two stochastic models for tournaments we use are the *Condorcet Random (CR) model* [143, 137] where $p(i,j) = 1-p$ if $i < j$ and p otherwise, and a *Linear* model where $p(i,j) = \frac{1}{2} + \frac{j-i}{2(N-1)}$. The CR model captures a situation where the comparator is wrong with some fixed probability, while the Linear model allows for the comparator to do better when the elements to be compared are significantly different. We used two settings of p for the CR model, $p = \frac{\log N}{N}$ and $p = \frac{1}{\sqrt{N}}$. Note that $p = \frac{\log N}{N}$ is a lower bound on the noise for the expected outdegree of the weakest player to be high enough to be able to win an SET. It has previously been shown that SETs generated with this model can be manipulated for most players [137].

We used all combinations of link function and model - practice CR, practice Linear, optimal Linear, optimal CR, SE CR and SE Linear, for all sizes between 8 and 1024 that

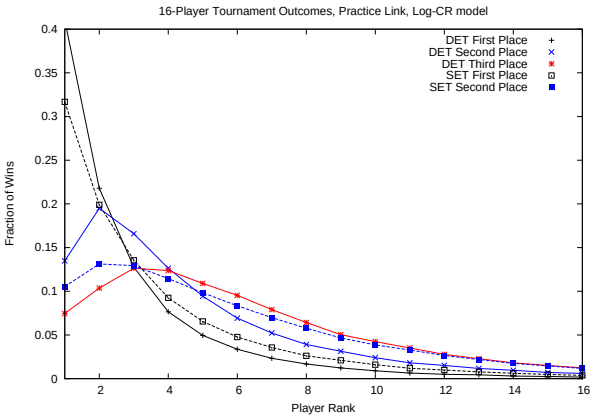

Figure 9.2: The distributions over first, second and third place for the 2 tournament constructions for 16 players with CR-Log noise.

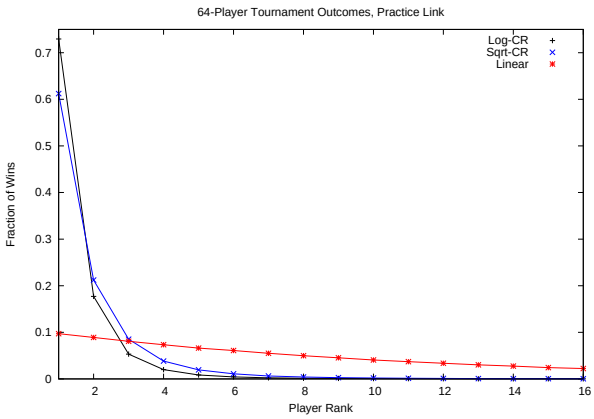

Figure 9.3: The distribution over players of first place with the link function in practice for the 3 noise models

are a power of 2. We increased the number of samples each time, starting with 50,000 for size 8, 100,000 for size 16, up to 6,400,000 for size 1024. For each run, we randomly sampled a seeding and then simulated each of the tournament models with the same seeding. We recorded the first, second and third place rank for each DE simulation and first and second for the SE simulations. The lower ranks are not well-defined for the outcome of DETs and SETs respectively.

Practice versus Optimal Link Function We found no significant difference between the distributions for first, second and third place between the two link functions. For each size tournament, we fixed the model and then calculated the total variation distance between the two distributions of equivalent size. The total variational distance was calculated as one half times the sum over all players of the absolute value of the probability they placed first with the practice link function minus the probability the same player placed first with the optimal link function. All results are reported in Table 9.1. In general, there was slightly more variation in the Linear model than the CR model, but none exceeded 0.7% change, and most were significantly smaller. Our conclusion from this is that improving the structure through using a better link function, for all intents and purposes, does not affect the outcome of DETs. Therefore, improving the link function by using the optimal link function is a net gain - it is fairer, more interesting for participants/observers, and just as effective.

Figure 9.3 demonstrates the distribution of players placing first, for the link function in practice. This shows how the results vary based on the model and noise parameter. As expected, the CR model with $\frac{\log N}{N}$ noise is more accurate that the one with $\frac{1}{\sqrt{N}}$ noise. Perhaps unexpectedly, the Linear noise model makes it extremely difficult to identify the top player. This becomes much more pronounced in the larger tournaments as the statistical advantage that players have over those ranked near them effectively disappears.

Rank	Model	8	16	32	64	128	256	512	1024
1	CR-Log	28.4	41.02	57.4	72.9	84.7	92.04	96.1	98.2
1	SET CRL	24.8	31.7	42.9	55.4	67.4	77.6	85.2	90.6
1	CR-Sqrt	31.4	40.7	51.1	61.2	70.6	78.6	85.03	89.8
1	SET CRS	26.8	31.4	37.8	44.9	52.3	59.6	66.6	72.8
1	Linear	44.4	27.7	16.5	9.7	5.6	3.1	1.72	0.9
1	SET Lin	39.7	24.1	14.4	8.5	4.9	2.8	1.5	0.8
2	CR-Log	17.1	19.5	29.5	45.6	63.2	77.8	87.7	93.5
2	SET CRL	14.8	13.1	14.9	19.3	25.3	31.6	37.2	41.6
2	CR-Sqrt	18.4	19.5	24.6	32.6	42.8	53.6	64.3	73.4
2	SET CRS	15.2	12.8	13.1	14.6	17.2	20.4	24.1	27.9
2	Linear	24.8	15.9	10.3	6.6	4.0	2.4	1.4	0.7
2	SET Lin	19.3	12.7	8.4	5.4	3.4	2.1	1.2	0.6
3	CR-Log	14.2	12.6	15.7	23.3	34.6	46.5	56.5	63.6
3	CR-Sqrt	14.7	12.5	13.4	16.1	20.7	26.9	34	41.2
3	Linear	18.6	11.4	7.64	5.1	3.2	1.9	1.2	0.6

Table 9.2: Percentage of simulations where the players ranked $\{1, 2, 3\}$ placed correctly

Improvement over SET results We found a large improvement in the results of DETs versus SETs, especially for larger sized tournaments and the CR model. In Figure 9.2, we have charted the distribution over the probability of each player placing first (black lines), second (blue lines) and third (red line) for both SE (dashed lines) and DETs (solid lines). Observe that DETs have a higher probability of the first player winning, and of the second player placing second than SETs. SETs do not have a defined third placed player, but the red line for DETs does place the most probability on the third and fourth ranked players. While we have only charted this for 16 players for clarity, the other distributions are all similarly shaped as can be seen in Table 9.2.

In Table 9.2 we give the the fraction of times the player ranked first, second or third by the model placed correctly in both DET and SET simulations. Note that the performance of the linear model decreases as the number of players increases, while we see the opposite with the CR model. This phenomenon can also be observed in Figure 9.3. While the largest improvements in performance appear for the second ranked player with 1024 contestants from 41.6% to 93.5% for log noise, and 27.9% to 73.4% for sqrt noise, there is always a noticeable difference between the SET and DET results.

9.5 Conclusions and Open Problems

In this chapter, we introduced formal definitions for the structure of double-elimination tournaments, following several natural design goals and mimicking the DETs used in practice. We showed that the link functions in practice are not optimal with respect to repeat-avoidance

and provided a constructive proof for creating optimal link functions with respect to repeat-avoidance.

We also presented several experimental results. We investigated the power of DETs for picking the winner in several stochastic generative models. The experiments showed that DETs are much better than SETs at identifying the strongest players in the natural stochastic tournament generating models that we used. Additionally, to investigate the proposed change in the link function, we tested whether the quality of the outcome is affected by the change. We founnd that, using total variational distance, the distributions generated are nearly identical. Therefore, the new proposed link function is strictly superior over the ones used in practice.

This work introduces several open questions. In addition to the ones stated in Section 3, another question raised by the identity link function is the relationship between winners in DETs and SETs. Given the same deterministic match outcomes, one can show that the set of SET winners is exactly those who can win a DET. What is this relationship for other link functions?

Bibliography

[1] `http://pywebgraph.sourceforge.net`, 2011.

[2] `http://facebook.com/press/info.php?statistics`, Jan 2012.

[3] `http://research.microsoft.com/ldg`, Jan 2012.

[4] `http://research.microsoft.com/trinity`, Jan 2012.

[5] M. Adler, P. Gemmell, M. Harchol-Balter, R. M. Karp, and C. Kenyon. Selection in the presence of noise: the design of playoff systems. In *Symposium on Discrete Algorithms*, pages 564–572, 1994.

[6] K. Jin Ahn and S. Guha. Graph sparsification in the semi-streaming model. *International Colloquium on Automata, Languages and Programming*, 2009.

[7] W. Aiello, F. R. K. Chung, and L. Lu. A random graph model for massive graphs. *Symposium on the Theory of Computing*, pages 171–180, 2000.

[8] W. Aiello, F. R. K. Chung, and L. Lu. A random graph model for power law graphs. *Experimental Math*, 10:53–66, 2000.

[9] R. Alberich, J. Miro-Julia, and F. Rossello. Marvel universe looks almost like a real social network. *arXiv*, 2002.

[10] A. Allen, D.M. Schwartz, and J.L. Punch. Tournament strategies in hearing aid selection. *Journal of Speech and Hearing Disorders*, 47:363–372, 1997.

[11] N. Alon. Ranking tournaments. *SIAM Journal on Discrete Mathematics*, 20(1):137–142, 2006.

[12] Y. Amanatidis, B. Green, and M. Mihail. Flexible models for complex networks. *Poster at Center for Algorithms Randomness and Computation*, 2008.

[13] L. A. N. Amaral, A. Scala, M. Barthelemy, and H.E. Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences*, 97:11149–11152, 2000.

[14] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *Foundations of Computer Science*, 2006.

[15] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *Symposium on Theory of Computing*, pages 235–244, 2009.

[16] K. Andreev and H. Racke. Balanced graph partitions. *Theory of Computing Systems*, 39:929–939, 2006.

[17] D. R. Appleton. May the best man win? *The Statistician*, 44(4):529–538, 1995.

[18] S. Arora, S. Rao, and U. Vazirani. Expander flows, geo -metric embeddings and graph partitioning. *Journal of the ACM*, 2009.

[19] K. Arrow. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58:328–346, 1950.

[20] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *Proceedings of Very Large Databases*, 4(3):173–184, 2010.

[21] M. Bailly-Bechet, C. Borgs, A. Braunstein, J. Chayes, A. Dagkessamanskaia, J.-M. Francois, and R. Zecchina. Finding undetected protein associations in cell signaling by belief propagation. *arxiv*, 2011.

[22] A-L Barabasi and R Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[23] S. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Supercomputing*, 1995.

[24] J. Bartholdi, C. Tovey, and M. Trick. The computational difficulty of manipulating an election. *Social Choice Welfare*, 6:227–241, 1989.

[25] J. Bartholdi, C. Tovey, and M. Trick. How hard is it to control an election? *Mathematical and Computer Modeling*, 16(8/9):27–40, 1992.

[26] R. Baumann, V. Matheson, and C. Howe. Anomalies in Tournament Design: The Madness of March Madness. *Journal of Quantitative Analysis in Sports*, 6(2):1–9, 2010.

[27] M. Bayati, J. H. Kim, and A. Saberi. A sequential algorithm for generating random graphs. *Algorithmica*, 58(4):860–910, 2010.

[28] Z. Bi, C. Faloutsos, and F. Korn. The dgx distribution for mining massive, skewed data. *KDD*, pages 17–26, 2001.

[29] J. Blitzstein and P. Diaconis. A sequential importance sampling algorithm for generating random graphs with prescribed degrees, 2006.

[30] B. Bollobás. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *European Journal on Combinatorics*, 1:311–316, 1980.

[31] B. Bollobás, O. Riordan, J. Spencer, and G. Tusndy. The degree sequence of a scale-free random graph process. *Random Structures & Algorithms*, 18 (3):279–290, 2001.

[32] M. Braverman and E. Mossel. Noisy sorting without resampling. In *Symposium on Discrete Algorithms*, pages 268–276, 2008.

[33] A. Z. Broder. How hard is it to marry at random? (on the approximation of the permanent). *Symposium on the Theory of Computing*, pages 50–58, 1986.

[34] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *ACM Symposium on Cloud Computing*, 2011.

[35] K. Chaudhuri, E. Halperin, S. Rao, and S. Zhou. A rigorous analysis of population stratification with limited data. In *Symposium on Discrete Algorithms*, pages 1046–1055, 2007.

[36] Robert Chen and F. K. Hwang. Stronger players win more balanced knockout tournaments. *Graphs and Combinatorics*, 4(1):95–99, 1988.

[37] J. Chew. tsh: tournament shell. *http://www.poslarchive.com/tsh*, 2012.

[38] F. K. Chung, S. Handjani, and D. Jungreis. Generalizations of polya's urn problem. *Annals of Combinatorics*, 2003.

[39] F. K. Chung and F. K. Hwang. Do stronger players win more knockout tournaments? *J. of the Amer. Statistical Assoc.*, 73:593–596, 1978.

[40] F. R. K. Chung and L. Lu. The average distance in a random graph with given expected degrees. *Internet Mathematics*, 1(1), 2003.

[41] Fan Chung, Linyuan Lu, , and Van Vu. Spectra of random graphs with given expected degrees. *Proceedings of the National Academy of Sciences*, 100:6313–6318, 2003.

[42] F.R.K. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *ANNALS OF COMBINATORICS*, pages 125–145, 2002.

[43] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[44] D. Coppersmith, L. Fleischer, and A. Rudra. Ordering by weighted number of wins gives a good ranking for weighted tournaments. In *Symposium on Discrete Algorithms*, pages 776–782, 2006.

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[46] E. Drinea, A. Frieze, and M. Mitzenmacher. Balls and bins models with feedback. In *SODA*, pages 308–315, 2002.

[47] Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *FOCS*, pages 673–682, 2010.

[48] E. Durant. Hearing aids and methods and apparatus for audio fitting thereof. (20100172524A1), July 2010.

[49] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research National Bureau of Standards Section*, 69:125–130, 1965.

[50] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[51] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[52] C. T. Edwards. Double-elimination tournaments: Counting and calculating. *The American Statistician*, 50(1):27–33, 1996.

[53] P. Erdős and A. Rényi. On random matrices. *Publications of the Mathematical Institute of the Hungarian Academy of Science*, 8:455–561, 1964.

[54] G. Even, J. Naor, S. Rao, and B. Schieber. Fast approximate graph partitioning algorithms. *SIAM Journal on Computing*, 28(6):2187–2214, 1999.

[55] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM*, pages 251–262, 1999.

[56] A. Flaxman, A. Frieze, and J. Vera. A geometric preferential attachment model of networks. *Workshop on Algorithms for the Webgraph*, pages 44–55, 2004.

[57] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[58] L.R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399404, 1956.

[59] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for general graph matching problems. *J. ACM*, 38(4):815–853, October 1991.

[60] T. Mc Garry and R.W. Schutz. Efficacy of traditional sport tournaments. *J. of Op. Research Society*, 48(1):65–74, 1997.

[61] A. Gibbard. Manipulation of voting schemes: a general result. *Econometrica*, 41, 1973.

[62] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99:7821–7826, 2002.

[63] C. Gkantsidis, M. Mihail, and E. W. Zegura. The markov chain simulation method for generating connected power law random graphs. *Meeting on Algorimic Engineering and Experimentation (ALENEX)*, pages 16–25, 2003.

[64] W. A. Glenn. A comparison of the effectiveness of tournaments. *Biometrika*, 47:253–262, 1960.

[65] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[66] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35:921–940, 1988.

[67] S. L. Hakimi. On the realizability of a set of integers as degrees of the vertices of a graph. *SIAM Journal on Applied Mathematics*, 10:496–506, 1962.

[68] V. Havel. A remark on the existence of finite graphs. *Caposis Pest. Mat.*, 80, 1955.

[69] N. Hazon, P.E. Dunne, S. Kraus, and M. Wooldridge. How to rig elections and competitions. In *COMSOC*, 2008.

[70] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing*, 1995.

[71] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. Physical Review Letters E, 2002.

[72] J. Horen and R. Riezman. Comparing draws for single elimination tournaments. *Operations Research*, 33(2):249–262, 1985.

[73] F. K. Hwang. New concepts in seeding knockout tournaments. *The American Mathematical Monthly*, 89(4):235–239, 1982.

[74] M. Jerrum and A. Sinclair. Fast uniform generation of regular graphs. *Theor. Comput. Sci.*, 73(1):91–100, 1990.

[75] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM*, 51(4):671–697, 2004.

[76] R. Kannan, P. Tetali, and S. Vempala. Simple markov-chain algorithms for generating bipartite graphs and tournaments. *Random Structures and Algorithms*, 14(4):293–308, 1999.

[77] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *International Conference on Parallel Processing*, pages 113–122, 1995.

[78] J. Kelner and A. Levin. Spectral sparsification in the semi-streaming setting. *Symposium on Theoretical Aspects of Computer Science*, pages 440–451, 2011.

[79] J. H. Kim and V. Vu. Generating random regular graphs. *Combinatorica*, 26(6):683–708, 2006.

[80] J. Kleinberg. Small-world phenomena and the dynamics of information. *Neural Information Processing Systems*, pages 431–438, 2001.

[81] D. E. Knuth. The stanford graphbase: A platform for combinatorial computing, 1993.

[82] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Random graph models for the web graph. *Foundations of Computer Science*, pages 57–65, 2000.

[83] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *World Wide Web (WWW)*, 2010.

[84] Lacrosse Outreach Foundation. Lacrosse jamboree rules. *http://www.lacrosseoutreach.org/Jamboree-Details/2011-boys-jamboree-rules.html*, 2012.

[85] J.A. Ladwig and N.C. Schwertman. Using probability and statistics to analyze tournament competitions. *Chance*, 5:49–53, 1992.

[86] J. Lang, M. S. Pini, F. Rossi, K. B. Venable, and T. Walsh. Winner determination in sequential majority voting. In *International Joint Conference on Artificial Intelligence*, pages 1372–1377, 2007.

[87] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

[88] J. R. Lee, S. O. Gharan, and L. Trevisan. Multi-way spectral partitioning and higher-order cheeger inequalities. In *Symposium on Theory of Computing*, pages 1117–1130, 2012.

[89] J. Leskovec. `http://snap.stanford.edu/snap`, 2012.

[90] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 2010.

[91] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. *KDD*, pages 177–187, 2005.

[92] D. Lita. Method and apparatus for managing billiard tournaments. (20080269925), Oct 2008.

[93] A. Louis, P. Raghavendra, P. Tetali, and S. Vempala. Many sparse cuts via higher eigenvalues. In *Symposium on Theory of Computing*, pages 1131–1140, 2012.

[94] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Uncertainty in Artifical Intelligence*, 2010.

[95] D. Lusseau, K. Schneider, O. J. Boisseau, P. Haase, E. Slooten, and S. M. Dawson. The emergent properties of a dolphin social network, 2003.

[96] P. Mahadevan, C. Hubble, D. V. Krioukov, B. Huffaker, and A. Vahdat. Orbis: rescaling degree correlations to generate annotated internet topologies. *SIGCOMM*, pages 325–336, 2007.

[97] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat. Systematic topology analysis and generation using degree correlations. *SIGCOMM*, pages 135–146, 2006.

[98] K. Makarychev, Y. Makarychev, and A. Vijayaraghavan. Approximation algorithms for semi-random partitioning problems. In *Symposium on Theory of Computing*, pages 367–384, 2012.

[99] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. Principles Of Distributed Computing, 2009.

[100] E. Marchand. On the comparison between standard and random knockout tournaments. *The Statistician*, 51:169–178, 2002.

[101] W. Maurer. On most effective tournament plans with fewer games than competitors. *Annals of Statistics*, 3:717–727, 1975.

[102] B. McKay. Asymptotics for symmetric 0-1 matrices with prescribed row sums. *Ars Combinatoria A*, 19:15–25, 1985.

[103] F. McSherry. Spectral partitioning of random graphs. In *Foundations of Computer Science*, pages 529–537, 2001.

[104] M. Mihail and C. Papadimitriou. On the eigenvalue power law. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, RANDOM '02, pages 254–262, London, UK, 2002. Springer-Verlag.

[105] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *ACM/USENIX Internet Measurement Conference*, 2007.

[106] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Proceedings of the 39th Annual Allerton Conference on Communication, Control, and Computing*, 2001.

[107] M. Mitzenmacher and E. Upfal. *Probably and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[108] M. Newman. Analysis of weighted networks. *Physical Review Letters E*, 70(5):056131, Nov 2004.

[109] M. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103:8577–82, 2006.

[110] M. E. J. Newman. The structure of scientific collaboration networks. *National Academy of Science*, 98:404–9, 2001.

[111] M. E. J. Newman. Assortative mixing in networks. *Physical Review Letters*, 89:208701, May 20 2002.

[112] M. E. J. Newman. Mixing patterns in networks. *Physical Review Letters E.*, 67:026126, February 04 2002.

[113] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices, 036104. *Physical Review Letters E*, 74, 2006.

[114] M. E. J. Newman. A symmetrized snapshot of the structure of the internet at the level of autonomous systems, reconstructed from bgp tables posted by the university of oregon route views project. *Unpublished*, July 22, 2006.

[115] Andrew Y. Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In T. G. Dietterich, S. Becker, and Zoubin Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, Cambridge, MA, 2002. MIT Press.

[116] Northern Virginia Football Club. Northern virginia opens championship tournament. *http://www.novafc.org/pdl/royals_super_20/index_E.html*, 2012.

[117] D. M. Pennock, G. Flake, S. Lawrence, E. J. Glover, and C.L. Andgiles. Winners dont take all: Characterizing the competition for links on the web. *Proceedings of the National Academy of Sciences*, 99:5207–5211, 2002.

[118] PlayPool.com.

[119] M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *RANDOM*, pages 159–170, 1998.

[120] A. E. Raftery and S. M. Lewis. The number of iterations, convergence diagnostics and generic metropolis algorithms. *Practical Markov Chain Monte Carlo*, pages 115–130, 1995.

[121] E. Ravasz and A. L Barabasi. Hierarchical organization in complex networks. *Physical Review Letters E*, 67:026112, 2003.

[122] A.E. Rosenberg. Effect of glottal pulse shape on the quality of natural vowels. *Journal of the Acoustical Society of America*, 49(2):583–590, 1971.

[123] S. Ross and S. Ghamami. Efficient simulation of a random knockout tournament. *Journal Industrial and Systems Engineering*, 2:88–96, 2008.

[124] T. Russell and T. Walsh. Manipulating tournaments in cup and round robin competitions. In *ADT*, pages 26–37, 2009.

[125] A. Sala, S. Gaito, G. P. Rossi, H. Zheng, and B. Y. Zhao. Revisiting degree distribution models for social graph analysis. *http://arxiv.org/abs/1108.0027*, 2011.

[126] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Sparse cut projections in graph streams. In *European Symposium on Algorithms*, pages 480–491, 2009.

[127] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. *Journal of the ACM*, 58(3):13, 2011.

[128] M. A. Satterthwaite. Strategy-proofness and arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions. *Journal of Economic Theory*, 10, 1975.

[129] D. T. Searls. On the probability of winning with different tournament procedures. *J. of the American Statistical Association*, 58:1064–1081, 1963.

[130] C. Seshadhri, T.G. Kolda, and A. Pinar. Community structure and scale-free collections of erdös-rényi graphs. *Physical Review Letters E*, 85:056109, May 2012.

[131] C. Sewell. SQBS, a quiz bowl statistics program. *http://ai.stanford.edu/~csewell/sqbs/*, 2012.

[132] A. Sinclair and M. Jerrum. Approximate counting, uniform generation and rapidly mixing markov chains. *Inf. Comput.*, 82(1):93–133, 1989.

[133] P. Slater. Inconsistencies in a schedule of paired comparisons. *Biometrika*, 48(3/4):303–312, 1961.

[134] A. Sokal. Monte carlo methods in statistical mechanics: Foundations and new algorithms. *Cours de Troisi'eme Cycle de la Physique en Suisse Romande, Lausanne*, 1996.

[135] Daniel A. Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. 2008.

[136] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *ACM Symposium on Knowledge Discovery and Data Mining*, 2012.

[137] I. Stanton and V. Vassilevska Williams. Manipulating stochastically generated single elimination tournaments for nearly all players. *WINE*, pages 326–337, 2011.

[138] I. Stanton and V. Vassilevska Williams. Rigging tournament brackets for weaker players. *IJCAI*, pages 357–364, 2011.

[139] A. Steger and N. C. Wormald. Generating random regular graphs quickly. *Combinatorics, Probability & Computing*, 8(4), 1999.

[140] R. Taylor. Constrained switching in graphs. *SIAM journal on algebraic and discrete methods*, 3, 1:115–121, 1982.

[141] Z. Toroczkai and K.E. Bassler. Network dynamics: Jamming is limited in scale-free systems. *Nature*, 428:716, 2004.

[142] United States Croquet Association. Draw and process format. *http://www.croquetamerica.com/croquet/tournaments/DrawAndProcess.php*, 2012.

[143] V. Vassilevska Williams. Fixing a tournament. In *AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 895–900, 2010.

[144] T. Vu, N. Nazon, A. Altman, S. Kraus, Y. Shoham, and M. Wooldridge. On the complexity of schedule control problems for knock-out tournaments. *JAIR*, 2010.

[145] T. Vu and Y. Shoham. Fair seedings in knockout tournaments. *ACM Transactions on Intelligent Systems and Technology*, 2010.

[146] T. Vu and Y. Shoham. Optimal seeding in knockout tournaments. In *AAMAS*, pages 1579–1580, 2010.

[147] C. Walshaw. `http://staffweb.cms.gre.ac.uk/~wc06/partition`, 2011.

[148] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[149] J. G. White, E. Southgate, J. N. Thompson, and S. Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Phil. Trans. R. Soc. London*, 314:1–340, 1986.

[150] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.

[151] W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33:452–473, 1977.

[152] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Technical Report UCB/EECS-2011-82*, July 2011.

[153] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[154] M. Zelke. Intractability of min- and max-cut in streaming graphs. *IPL*, 111(3):145 – 150, 2011.