Spring 5-2020

HEURISTICS FOR SPARSEST CUT APPROXIMATIONS IN NETWORK FLOW APPLICATIONS

Fernando Vilas
fernando.e.vilas@gmail.com

HEURISTICS FOR SPARSEST CUT APPROXIMATIONS
IN NETWORK FLOW APPLICATIONS

Approved by:

Dr. David W. Matula
Professor Emeritus - CS (Advisor)

Dr. Eli Olinick
Associate Professor - EMIS

Dr. Sukumaran Nair
University Distinguished Professor - ECE

Dr. Michael Hahsler
Associate Professor - CS

Dr. Erik Larson
Associate Professor - CS

Dr. Jeff Tian
Professor - CS (Committee Chair)

HEURISTICS FOR SPARSEST CUT APPROXIMATIONS IN NETWORK FLOW APPLICATIONS

A Dissertation Presented to the Graduate Faculty of the

Bobby B. Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Science

by

Fernando E Vilas

B.S., Louisiana State University, 2002
M.B.A., Louisiana State University, 2004
M.S., Southern Methodist University, 2010

May 16, 2020

ACKNOWLEDGMENTS

I would like to acknowledge the support of several groups without whom this would not have been possible. First, my company who paid for my advanced education and made schedule accommodations for me to finish. Next, my family and friends who pushed me to continue and were understanding when school had to take priority over other things. Then, my committee for their advice on classes, conferences, papers, and research. And finally, the SMU support staff, especially Beth Minton, who makes the department function and helped me with all the administrative paperwork over the years so that the research could continue.

Vilas, Fernando E

B.S., Louisiana State University, 2002
M.B.A., Louisiana State University, 2004
M.S., Southern Methodist University, 2010

Heuristics for sparsest cut approximations
in network flow applications

Advisor: Dr. David W. Matula
Doctor of Philosophy degree conferred May 16, 2020
Dissertation completed May 8, 2020

The Maximum Concurrent Flow Problem (MCFP) is a polynomially bounded problem that has been used over the years in a variety of applications. Sometimes it is used to attempt to find the Sparsest Cut, an NP-hard problem, and other times to find communities in Social Network Analysis (SNA) in its hierarchical formulation, the HMCFP. Though it is polynomially bounded, the MCFP quickly grows in space utilization, rendering it useful on only small problems. When it was defined, only a few hundred nodes could be solved, where a few decades later, graphs of one to two thousand nodes can still be too much for modern commodity hardware to handle.

This dissertation covers three approaches to heuristics to the MCFP that run significantly faster in practice than the LP formulation with far less memory utilization. The first two approaches are based on the Maximum Adjacency Search (MAS) and apply to both the MCFP and the HMCFP used for community detection. We compare the three approaches to the LP performance in terms of accuracy, runtime, and memory utilization on several classes of synthetic graphs representing potential real-world applications. We find that the heuristics are often correct, and run using orders of magnitude less memory and time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acronyms

MCFP Maximum Concurrent Flow Problem. 1

OOM Out of Memory. 18, 19

SCP Sparsest Cut Problem. 1

SNAP Stanford Large Network Dataset Collection. 18, *Glossary:* Stanford Large Network Dataset Collection

cut density Given a graph partitioned into two parts, S and T, that cuts through $\sum w_e$ total weight on the cut edges, the cut density is defined as

$$density = \frac{\sum w_e}{|S| * |T|}$$

. 2

giant component The largest connected component in a graph. 18

graph diameter The longest of all the pairwise shortest paths in a graph. 9

k-cut density The value of the cut density when the graph is partitioned into k parts, shown as $A_i, 1 \leq i \leq k$. This is given as

$$density(A_1, A_2, A_3, ..., A_n) = min_w \frac{\sum_{1 \leq i \leq j \leq k} w_{ij} C(A_i, A_j)}{\sum_{1 \leq i \leq j \leq k} w_{ij} D(A_i, A_j)}$$

where w_{ij} is the weight on the edge (i, j), C is the capacity function on all edges, and D is the demand function between all pairs. So $C(A_i, A_j)$ represents the total capacity on all edges between partition A_i and partition A_j, and similarly $D(A_i, A_j)$ represents the demand between all pairs with one end in A_i and the other in A_j. 2

modularity A measure in the range [0,1] of the community structure in a graph, with 0 being random and higher values implying more structure. It is defined as $Q = \sum_i (a_{ii} - b_i^2)$. With a_{ii} as the fraction of edges in community i and b_i as the fraction of all edges originating in community i that end in another community. 23

power law A relation where one value is proportional to an exponential change in the other, i.e. $x \propto y^n$. 21

residual capacity The amount of unused capacity in a slack edge that is not being used to transport flow across the critical edges from a node that is not adjacent to a critical edge. 31

slack edge An edge that is not a critical edge, or slack in the capacity constraint, therefore having residual capacity. 31

standard deviation A measure of dispersion of a set of N values. Higher implies more dispersion from the mean μ of the set. It is often represented as σ and defined as

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$$

. 20

Stanford Large Network Dataset Collection A collection of large real-world network datasets available at https://snap.stanford.edu/data. xii, 18

triangle inequality Given three nodes A, B, and C, and weights w on the edges between the nodes, $w_{AB} + w_{BC} \geq w_{AC}$. An example of a weighting function that might not obey the triangle inequality is travel time when considering traffic congestion. 2

This is dedicated to my wife Candace, who supported the crazy schedule of graduate school with a day job, and finished two degrees of her own while I was working on this one.

Chapter 1

INTRODUCTION

This document explores several approaches to the related problems of sparsest cuts and maximum concurrent flows within the context of a few interesting applications. The problems of determining the Sparsest Cut (SCP) in a graph and the Maximum Concurrent Flow (MCFP) in a network have been extensively studied since their introduction by Shahrhoki and Matula in [98]. While sparse cuts are interesting for several reasons, the MCFP has proven to be an interesting problem in its own right.

1.1. Scoping the Problem

In order to effectively study the area in a timely manner, our studies are limited to three heuristics and how well they perform relative to an exact solution. There is an additional excursion into quantum computing that showed similar results to those of a classical computer in Chapter 7.

The performance of the heuristics is evaluated on several classes of graphs, both real and synthetic, that represent reasonable inputs from several potential areas of application.

1.1.1. Sparsest Cut

The Sparsest Cut Problem (SCP) was first presented as a weak dual of the Maximum Concurrent Flow Problem (MCFP) in [112] which was previously described as a method of average linkage hierarchical clustering in [94].

The sparsest cut problem is similar to the minimum cut problem, which seeks the smallest number of edges to cut that disconnects the graph, or the minimum weight edge set in a weighted graph. It has been shown that the minimum cut problem is solvable in polynomial time with the Edmonds-Karp algorithm [44]. However, the sparsest cut problem seeks a

collection of edges that constitute a cut of minimum density, rather than minimum weight. It is this nuance that makes the problem NP-complete [112].

It is also possible that there are several cuts of similar density that, if considered together, would partition the graph into multiple parts. This is called a k-partite cut or a k-cut for short [54], which has been explored in [61]. In the case of the sparsest k-cuts, the k-cut density is given in [112] as follows:

$$den(A_1, A_2, A_3, ..., A_n) = min_w \frac{\sum_{1 \leq i \leq j \leq k} w_{ij} C(A_i, A_j)}{\sum_{1 \leq i \leq j \leq k} w_{ij} D(A_i, A_j)}$$

for any non-zero distance function w that obeys the triangle inequality.

The solution to the sparsest cut problem has been used in fields such as security attack graphs [127] and finding Nash equilibria in games [12].

The cut density of an (S, T)-cut in an undirected graph $G = (V, E)$ is $|(S, T)|/(|S| * |T|)$ where $|(S, T)|$ is the number of edges between node sets $S \subset V$ and $T \subset V$, and $|S| * |T|$ is the maximum number of edges possible (or separated demand). A minimum density cut in the graph is a sparsest cut. When the edges are weighted, the density is the average weight of the (S, T)-cut edges, with absent edges treated as edges of zero weight. The sparest cut problem is NP-hard [57, 98], and so it is unlikely that it can be solved in all cases with an efficient (polynomial time) algorithm. Further, the SCP was shown to be APX-hard, that is any approximation of it within a constant factor will also be NP-hard. However, the sparsest cut problem is closely related to the MCFP by $z^* \leq \mu^*$ where z^* is the maximum concurrent flow and μ^* is the minimum cut density in the SCP. This relationship can be exploited in many cases, since the MCFP can be solved in polynomial time via linear programming (LP).

The famous result from Ford and Fulkerson [49,50], the max-flow/min-cut theorem, states that the maximum flow between a given source node s and sink node t in a graph is equal to the capacity of a minimum capacity cut separating s and t. Similarly, it can be shown that the value of the maximum concurrent flow (MCF) in a graph is less than or equal to the density of a sparsest cut. More significantly, when the MCF solution identifies a partition into two to four parts, a sparsest cut can be identified [93, 98], giving a strong dual relationship.

Similarly, it can be shown that the two to four parts are precisely the components obtained by removing the edges of all sparsest cuts. Further, in [81] it was shown that for the weak dual cases, the MCF is within $O(\log n)$ of the minimum cut density.

1.1.2. Maximum Concurrent Flow Problem (MCFP)

An important problem in processing sensor data is that of finding structure in the results. Modern sensors and sensor networks can generate large volumes of data that must be quickly transformed into actionable information, or discarded to make room for new data. Previous attempts at finding structure have often focused on clustering algorithms, including the use of sparse cuts to provide an average linkage clustering shown in [94].

The maximum concurrent flow problem (MCFP) is a peer-to-peer network flow problem defined on an edge-capacitated graph in which the objective function is to maximize the ratio of the flow delivered between each peer-to-peer pair in comparison to the corresponding demand for that pair. This ratio must be the same for all pairs and is known as the throughput of the concurrent flow [112]. The MCFP has applications to VLSI circuit design [31], and clustering and classification in biological taxonomy [55, 92, 94], and social network analysis (SNA) [91] as a result of its relationship to the sparsest cut problem [81, 98, 112]. It can be shown that every MCFP instance has a set of critical edges that are saturated with flow by every optimal solution [93].

Mann et al. [91] proposed the MCF Cut Algorithm for community detection that iteratively partitions a social network (graph) by removing edges corresponding to *sparse cuts* between successively denser communities (subgraphs). The authors establish conditions for the algorithm assuring that the densities increase monotonically with each cut and report the application of the algorithm to real-world networks with traditionally accepted community structures to evaluate how effectively the algorithm agrees with the accepted hierarchical community structures and randomly generated networks that have embedded community structures. The algorithm compares favorably with the results of the widely cited and well structured divisive Girvan-Newman algorithm [55] in finding the community structure of

dense, weighted social networks. The MCF Cut Algorithm is based on exploiting the duality between sparse cuts and maximum concurrent flow in graphs.

The Maximum Concurrent Flow Problem was introduced in [112] to represent the amount of throughput possible within a connected network. Given a graph of capacitated edges and demands between pairs of nodes, create a parameter z as a coefficient to all demands in the graph. The problem is to maximize z subject to all the capacity constraints. There are three different formulations fo the MCFP covered in Chapter 3.

The maximum concurrent flow problem (MCFP) is a computationally challenging network flow problem with applications in transportation, telecommunications, VLSI circuit design, and data clustering for biological taxonomy and social network analysis (SNA). The best known algorithms for solving general linear programming (LP) problems have recently improved from the $O(n^{3.5})$ given in [72], where n is the number of variables to $O(n^{2.38})$ in [37, 80]. Further, the triples formulation, the most efficient currently known [41] requires $O(|V|^3|E|)$ space. Some authors have approached the problem as one of max-min fair flow, especially when viewing maximum concurrent flow as a "water filling" algorithm [79, 100].The problem also does not lend itself well to parellelization. These combined issues limit the feasibility of the MCFP to comparatively small problems of a few hundred nodes.

The MCFP is often presented as a unit-capacity, unit-demand case, though most approaches to solving it work in cases with non-uniform capacity. Some approaches that work on the non-uniform demand case use two graphs: a capacity graph and a demand graph, similar to the approach taken by [57], illustrated by an example from [102]. The use of two graphs is discussed in Chapter 5 on graph coarsening.

An alternative formulation of the MCFP is the Minimum Capacity Utilization Problem (MCUP), also defined in [112]. In this case, the same graph as the MCFP is used. Instead of being bounded by capacity constraints, the demands must be met. Now a parameter u to be minimized as a coefficient of capacities. The parameters u and z are related to each other as $u = 1/z$. It can be useful to take advantage of this relationship in developing some heuristics.

Another relationship of note is that the sparsest cut problem is a weak dual of the MCFP in that the maximum concurrent flow is bounded from above by the sparsest cut. That is, $min(den) \geq z$. It was shown in [9] that if Khot's Unique Games Conjecture [75, 76] is true, then the MCFP formulation is at best a $O(\log n)$ approximation to the sparsest cut.

Graphs that have $min(den) = z$ are called "bottleneck graphs". Any graph that is partitioned by sparsest cuts into less than five components is guaranteed to be a bottleneck graph [98].

Any edges where the capacity is completely utilized by every optimal concurrent flow are called "critical edges". The removal of critical edges generates a partition of the graph into at least two parts, which can be used for divisive average linkage clustering.

Although it is computationally polynomial, the MCFP is a challenging problem because, "[it] gives rise to extremely difficult linear programs instances of a size and type relevant to applications often prove beyond the reach of state-of-the-art linear programming codes" [18]. While polynomially bounded, the MCF problem is "memory-hard", meaning that in practice, it requires asymptotically more space than processing power [105]. Furthermore, there is no known combinatorial algorithm for the MCFP except for the special case of a single supply node [16]. In prior art formulations, the MCFP is most naturally characterized by the edge-path formulation given in Chapter 3. Here all paths may have assigned flows and each edge constrains the total flow assigned to all paths going through that edge. While this model is natural and provides an intuitive description of the problem, it suffers from a linear programming (LP) formulation that grows exponentially with problem size. The equivalent node-edge MCFP formulation is a multicommodity flow formulation where, for bookkeeping purposes, each node designates a distinct commodity yielding a polynomially bounded LP formulation. The artificial distinction of separate commodity flows is unnecessary for many applications. The triples formulation given in [41] and discussed in more detail in Chapter 3 reduces the size of the problem, but still yields a large problem for which heuristic approaches are useful.

1.1.3. Hierarchical Formulations

With the above two problems, they provide the first layer of the data. Many other interesting problems involve the n-th layer of those. Mann [90] used a variant of the Hierarchical Maximum Concurrent Flow problem (HMCFP), providing a framework for finding community structure in social networks with better accuracy in the related centrality measure than previous work. Conceptually, the HMCFP is an iterative approach to the MCFP where the z values from each iteration are stored with the demand pairs separate by the associated cut. The approach from Mann was divisive in nature, allowing each iteration to solve a smaller problem. The approach in this text is more hierarchical in that it solves the same size problem at each iteration, with certain flows fixed based on the previous iteration. The definition of the HMCFP is formalized in Chapter 3.

1.2. Practical Applications

As mentioned previously, there are numerous applications for the MCFP and the SCP. Solutions for the MCFP have been used for VLSI floor planning [31], analysis of wireless sensor networks, and other applications. The social networking analysis described in [91] and [84] that use the HMCFP are of particular interest and described in more detail in below.

Sparse cuts have been used in the analysis of security attack graphs [127] and refining the solution space when multiple solutions are valid for underdetermined systems of equations. For instance, the field of Compressed Sensing relies on using sparse cuts to select sparse solutions to underdetermined systems of linear equations. Both the SNA and Compressed Sensing applications are covered in depth in Chapter 9.

1.2.1. Social Networking Analysis

An important problem in SNA is to describe the *community structure* of a given network. This involves identifying a hierarchy of clusters of actors in a social network who share common relationships or interactions. Social scientists often employ density measures to

find community structure in social networks [24, 111]. The density of a graph with n nodes and m edges is defined as $\frac{2m}{n^2-n}$, which is the ratio of m compared to the number of edges in a complete graph on n nodes, K_n.

The use of hierarchical sparse cuts to reveal community structure was considered in [91]. The results of the HMCFP were used to show cuts in small social networks, and shown as dendrograms. Later, in [90], the "flowthrough centrality" measure was introduced. It was also shown that the new centrality measure was significantly more robust to input perturbations than traditional centrality measures, such as betweenness.

Due to the robustness of the flowthrough centrality result to variation in the inputs, we hope to show that a heuristic for the HMCFP can provide useful results on large networks. That is, we show that the MAS heuristic provides "good enough" community identification for further analysis to be useful. Further, it can provide these results far faster than LP approaches and can handle larger graphs, due to the lower memory requirements.

1.2.2. Compressed Sensing

The field of Compressed Sensing is related to extracting sparse information from a large set of data. Based on past information and a current measurement, a set of linear equations is created and solved. Network flows can be used to find a set of allowable solutions, where the sparsest solution (the solution with the fewest non-zero coefficients) is the correct one with high probability when the problem obeys certain structure. The result is included in future iterations as a cutting plane for refining the estimator of the actual solution. [15]

Rather than solving the NP-hard problem of finding the sparsest solution, a heuristic is used to estimate the answer. The current algorithms in use are modifications of the ARV algorithm presented in [11]. While ARV is more efficient in the asymptotic case, it remains to be seen if our heuristics are close enough to the right answer and faster in the practice. It also provides a use case where the hierarchical cases are not needed, in contrast to the other applications like SNA mentioned here.

1.2.3. Wireless Sensor Networks

Wireless sensor networks have become more interesting as the number of wireless devices have proliferated. Small sensors and mesh networks create a graph that can be analyzed for total throughput between all devices. Previous work has focused on backbone identification for network longevity due to battery life issues [32, 39, 89]. These backbones also provide a set of graphs for throughput analysis using the MCFP or similar tools.

1.3. Current Approaches

The MCFP has so far only been formulated as an LP. In recent work, such as [41], new LP formulations are being developed for efficiency. The triples formulation was developed around the concept of flow diversion from saturated paths to less saturated ones. This new formulation reduces the size of the constraint matrix, though still requires $O(|V|^5)$ storage.

The flow diversion approach was also used in [19] with exponential toll functions to change the costs of routes so that least cost routes approximately align with the MCFP. A similar approach, but applied to the triples concept, should allow a heuristic solution to the MCFP for large problems. A rerouting approach based on the triples concept is a candidate for future work.

The approach used for an exact solution to the all-pairs minimum cut problem in [117] was extended by us to approximate the Sparsest Cut Problem. In most cases, this approach finds the correct solution, though on dense graphs, it can fail to detect the correct answer. We discuss this approach further and refine the findings in more detail in Chapter 4.

Other than the LP formulations, current approaches seem to be limited to finding the first cut (k-cut) in the graph. We have extended the MAS heuristic approaches to the hierarchical case. This includes investigating the errors when compared to LP solutions to the HMCFP in small graphs.

1.4. Methods and Contributions

This dissertation makes some specific contributions on heuristics for solving the MCFP and HMCFP. Chapter 4 covers using the MAS to traverse a graph in a manner that tends to stay within a community before moving to another community. Combining this with a visitor pattern to track cut density, we can find a sparse cut (but maybe not the sparsest) significantly faster than solving the LP. The approach also covers a method of extending the calculation to the hierarchical formulation. We show that this heuristic often finds the sparsest cut on the graphs analyzed and provide an analysis of the correctness of the heuristic in various conditions.

Chapter 5 extends the MAS approach to coarsen graphs into communities detected with that method, and then solve the resulting graph with an LP. As in the original work, the results are compared to solving the MCFP directly for runtime, memory utilization, and accuracy. This chapter also has specific examples of the coarsened graphs to illustrate the effect more clearly to the reader.

The third contribution is with the introduction of the D_3 bound in Chapter 6. This bound assumes a graph diameter of three to calculate a sparsest cut bound on the graph without having to execute an all-pairs shortest path algorithm. This approach is attempted on numerous random bipartite graphs of low diameter, with some greater than three. This result is compared to two other bounds on the sparsest cut, one from removing a node of minimum degree and the shortest path bound calculated as a function of all pairwise shortest paths. The D_3 bound compares favorably to the shortest path bound, even when the diameter is greater than three.

The remainder of the document is organized as follows: Chapter 2 goes into detail on the graphs used to evaluate the heuristics in the rest of the studies and their sources. Chapter 3 covers multiple LP formulations of the MCFP and HMCFP and provides an overview of LP algorithms with some considerations for reduction to practice in various libraries. Then Chapters 4, 5, and 6 explore the heuristics as the main thrust of the dissertation, as discussed in the above paragraphs. Chapter 7 uses quantum computing techniques to explore a random

walk to find the sparsest cut. Since the quantum coin is observed at each step, it is not a true quantum walk, however the chapter is a starting point for implementing a quantum diffusion in the future. Since the analysis of heuristic performance includes memory and processor utilization, it is important to have a constistent computing environment on all the tests. In Chapter 8 this computing environment is discussed, including the hardware used along with the libraries and the rationale behind using each library. Chapter 9 discusses some potential application areas where these heuristics may show value in the speed or memory usage compared to current approaches. Finally, Chapter 10 summarizes the dissertation and outlines several areas for future work in theoretical or empirical analysis and also in reduction to practice for application.

Chapter 2

DATA SOURCES

This chapter covers the various classes of test graphs used in the analysis in the rest of the document. It begins with some small test graphs from previous work to give continuity from those efforts and simplify debugging. Then there is a discussion on popular graphs from social networking literature, including the Florentine Families graph, the 2004-2006 NFL graph, and 2004-2006 NCAA graph, among others, to examine behavior of the heuristics in these cases. Finally, this chapter covers randomly generated graph classes, including random graphs, random geometric graphs, random bipartite graphs, and random typing graphs to provide a large corpus of synthetic graphs with different properties to continue the research without worrying about statistical bias.

2.1. Test Graphs From MCFP Literature

Early MCFP literature from Biswas, Thompson, and Matula introduced eight test graphs [19, 118] that we reuse here to examine the algorithms presented in the rest of this study. They are small enough to solve by hand, for checking algorithm performance.

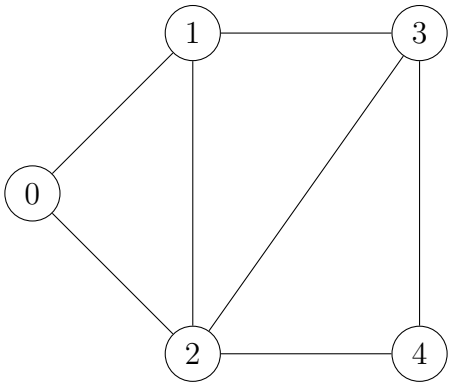


Figure 2.1. G1

G1 is the trestle graph on five nodes, where several cuts tie, but the graph is not gridlocked. The MCFP triples and node-edge formulations find enough cuts to separate the graph into five parts at the same level of the hierarchy. For some graphs, including G1, repeated iterations of the hierarchical form are needed to find all of the cuts at that level. We consider these to be the same cut since they are all ties, and the iteration is simply necessary to find all the ties. If the graph where gridlocked, the tied edges would have been found as a single cut.

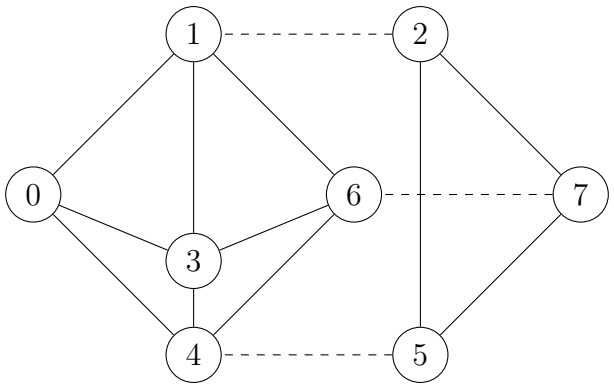


Figure 2.2. G2

G2 is an example of a dumbbell graph similar to those discussed in [94] that has a dense community of five nodes on the left and K_3 on the right, connected by three edges, shown as dashed lines. The sparsest cut and the MCFP critical edges separate these two communities.

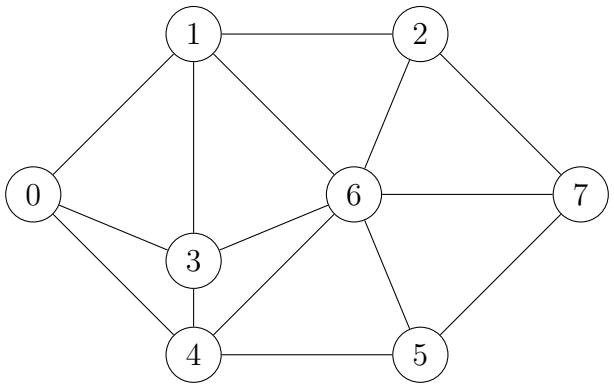


Figure 2.3. G3

G3 is a variation on G2, where the dumbbell structure is collaped into a central node of degree six. This change in structure has a significant impact on the optimal flows found using the MCFP. More flow overall is allowed between the nodes, and five edges are included in the sparsest cut, rather than three in G2.

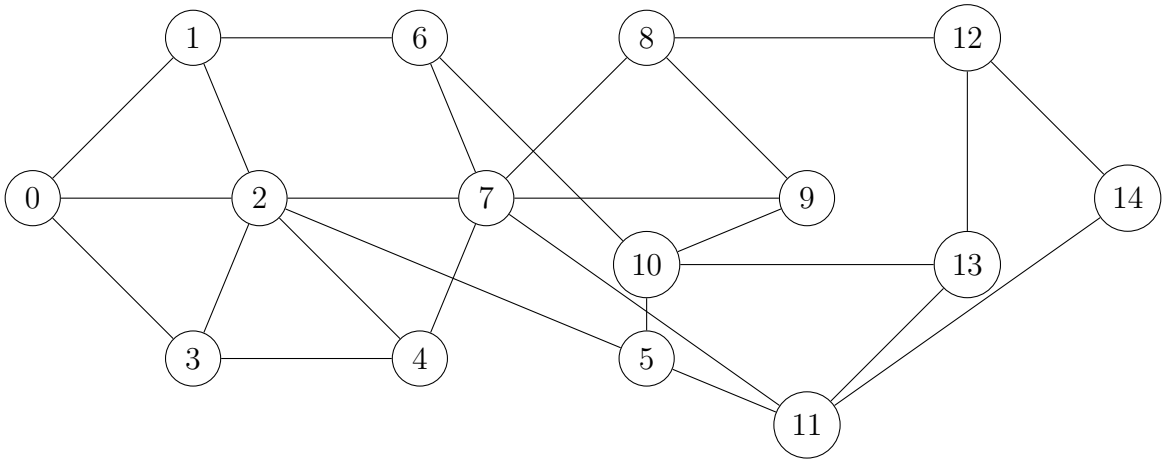


Figure 2.4. G4

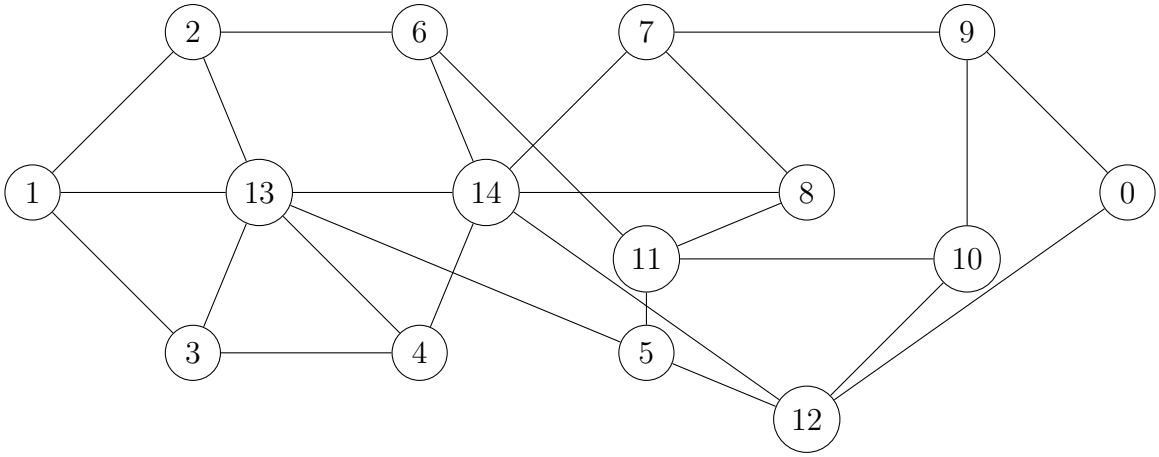


Figure 2.5. G5

G4 and G5 are isomorphic to each other. This provides a test of whether the order of the nodes matters when creating the LP or examining the performance of any heuristic.

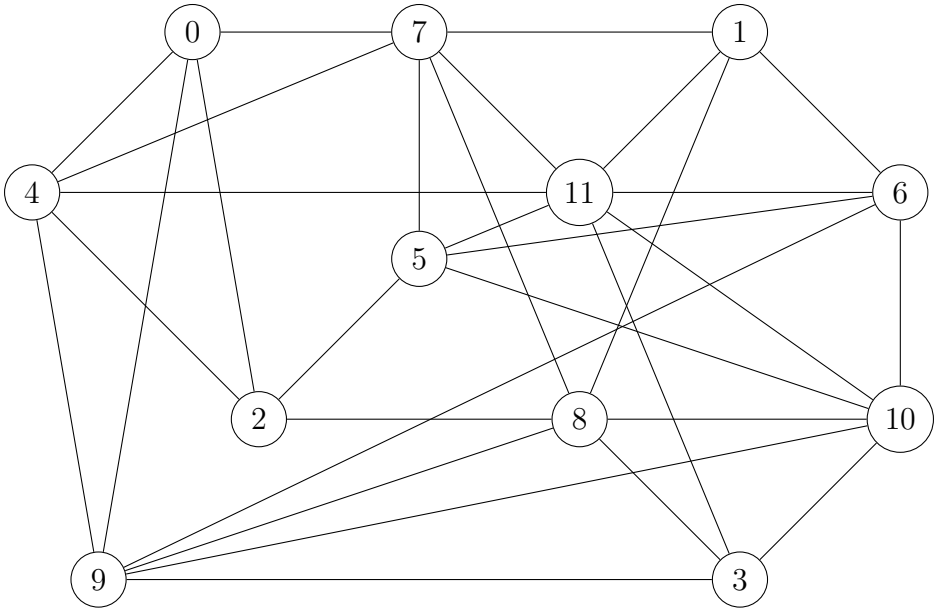


Figure 2.6. G6

G6 has the same MCFP throughput as the solution to the SCP. However, in [19], the rerouting heuristic separated node 2 for a throughput of $\frac{4}{11} = 0.3636$ rather than a sparsest cut of $\frac{7}{27} = 0.2592$ by separating nodes 0, 4, and 2. The MCFP itself finds the correct solution, as does the MAS heuristic.

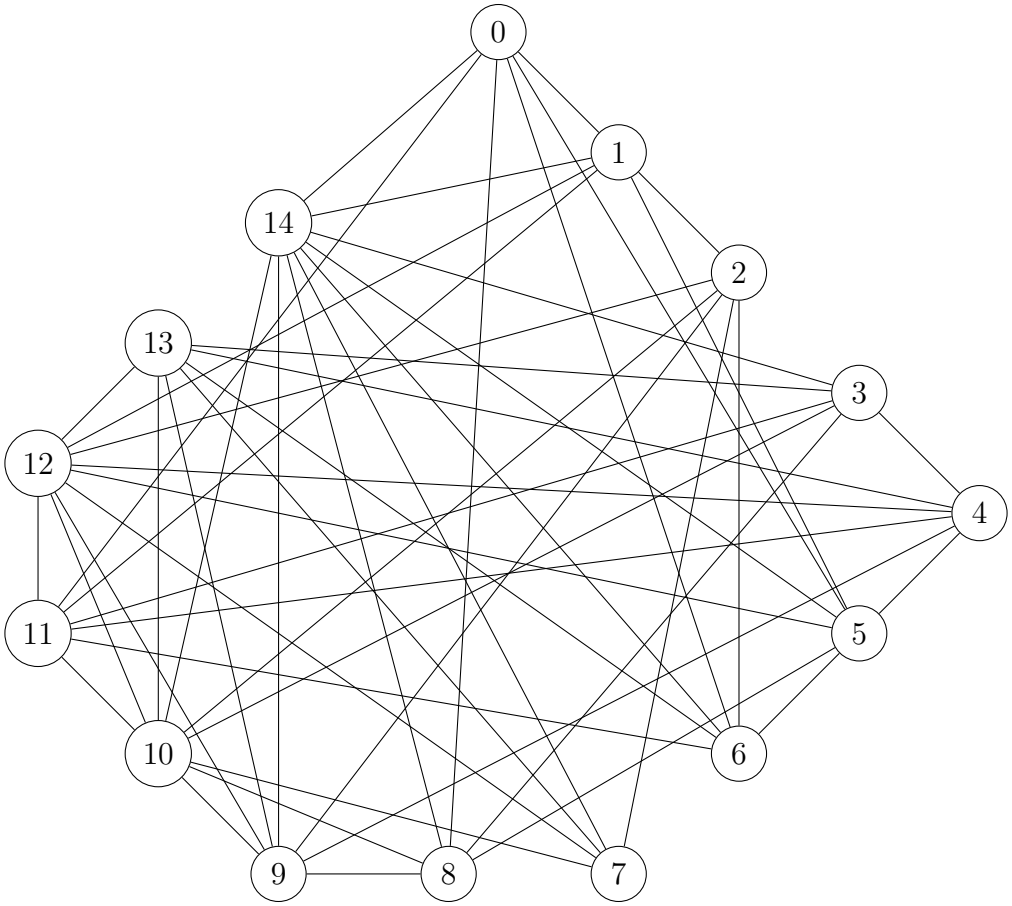


Figure 2.7. G7

G7 is a "near random" graph created to have diameter two, resulting in a high density. It was created by an early random graph generator by [118] that was designed to create random graphs that had certain properies to support different types of graph analysis. This type of graph is challenging for some heuristics such as flow swapping. However, our MAS heuristic, presented in Chapter 4 finds the same answer as the triples LP formulation of the MCFP.

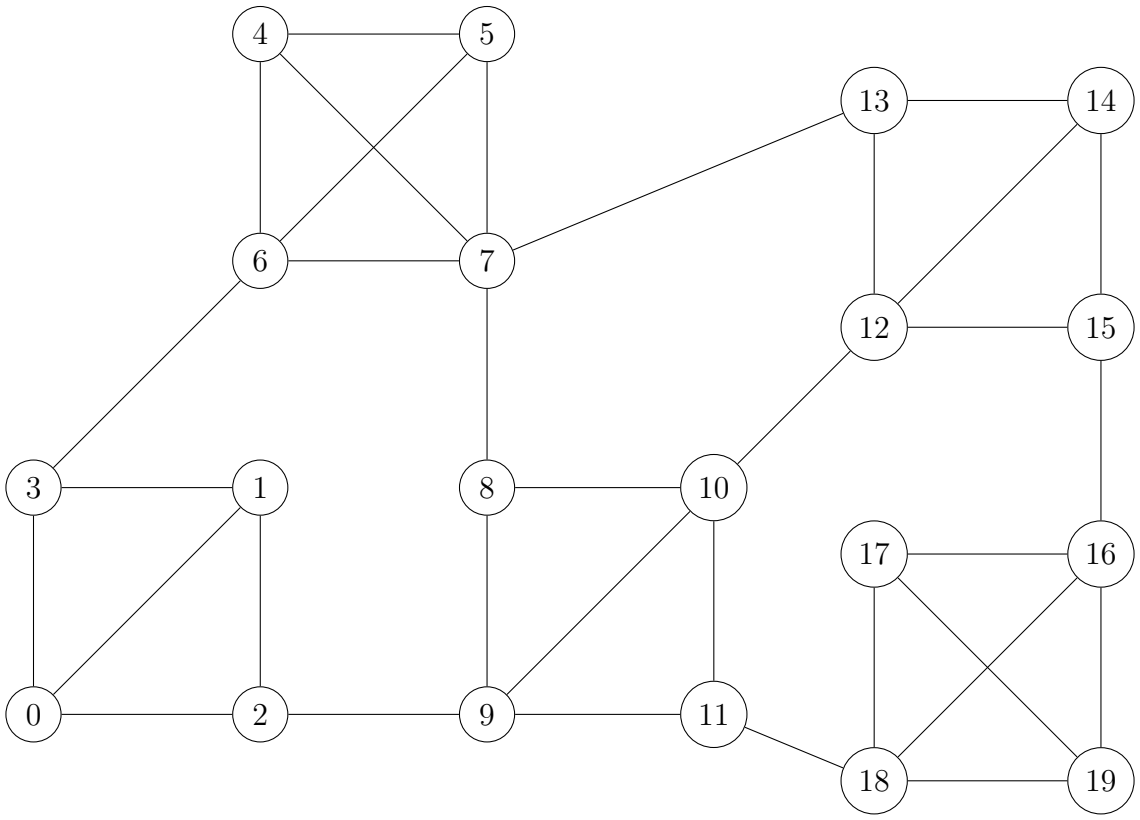


Figure 2.8. G8

G8 is a variation on the trestle graph in G1, with each node in G1 expanded to four nodes that form a community, for a total of 20 nodes. This graph is useful for examining coarsening approaches, since a good coarsening algorithm should collapse each of the 4-node communities into a single supernode, resulting in G1.

2.2. Test Graphs From Social Networking Literature

Social Networking Literature provides many sample graphs based on real world phenomena. These graphs provide a source of known performance for other approaches. We use some of these graphs, discussed in more detail below, to evaluate the performance of our approaches. However, there are not enough SNA graphs of similar types to provide statistical data on different approaches. For that, we use synthetic graphs, discussed in later sections.

2.2.1. Florentine Families

This Florentine Families graph is created from the Medici era marriages described in Machiavelli's The Prince [88]. It is a relatively small example of a real social network, with some interesting properties. The graph is also small enough to calculate algorithmic results by hand, on paper, for testing ideas before implementation on larger graphs.

2.2.2. NFL and NCAA Data

In his work on community detection, Mann used two edge-weighted graphs of interest [90]. For the NFL (Figure 2.9) and the NCAA Division 1 (Figure 2.10) football teams, he listed the teams and used the number of games played between pairs of teams from 2004-2006 as the capacity on the edges, while maintaining unit demand between all pairs. He did this for the 32 teams in the NFL and the 120 teams in the NCAA. In each of these cases, the graphs are large enough to be difficult to solve by hand, but solvable as an LP in a few minutes on a modern computer. They also represent two naturally occurring networks with a known structure for community detection algorithms.

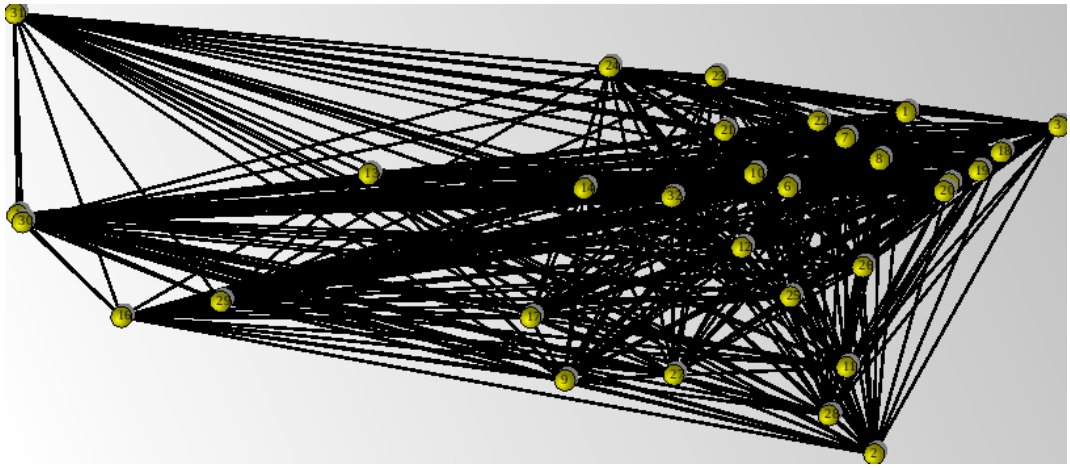


Figure 2.9. NFL 2004-2006 data

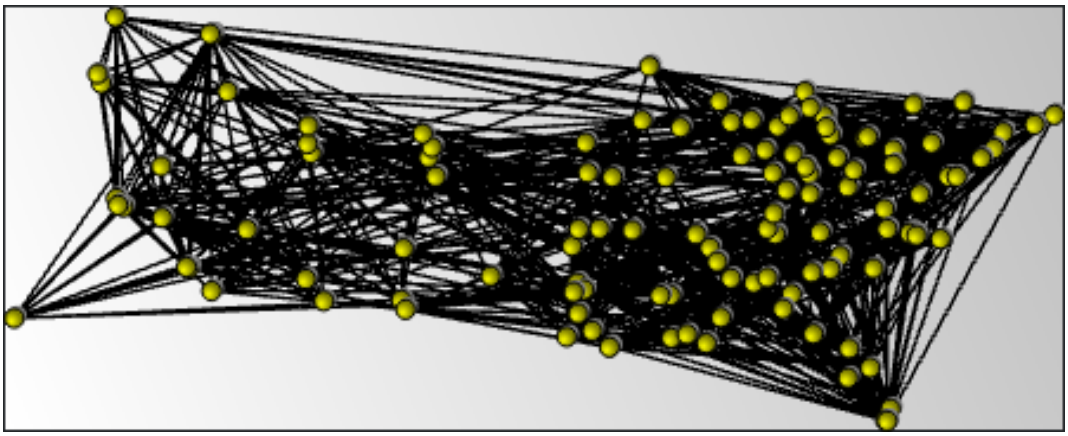


Figure 2.10. NCAA 2004-2006 data

2.2.3. Large Social Network Graphs

In the course of developing this body of work, multiple referees suggested operating on larger graphs to show relevance to the way SNA has grown to examine large social networks such as those on Twitter, Facebook, or Bitcoin. The growth of the problem makes the full MCFP LP formulation challenging to solve on graphs of only a few hundred nodes. Some smaller graphs are available through the Stanford Large Network Dataset Collection (SNAP) [83]. Even so, these graphs are far too large to solve on commodity hardware, though performance of heuristics and coarsening algorithms is worth evaluating. Most of the smaller SNAP graphs are still too large at several thousand nodes to evaluate our heuristics on current hardware. Future work might include updating the data structures in use so that these and larger graphs can be evaluated.

For example, we evaluated the EU core email graph [82, 126] that has a giant component of 986 nodes and 16,064 edges. The triples LP formulation in GLPK was over 30GB trying to create the model after reading the data file. After one hour, the Out of Memory (OOM) killer killed the process. However, the MAS heuristic solved the graph in a few seconds with $z = 0.00101523$. The coarsening approach reduced the problem space to 908 nodes and 9,510 edges. This created a problem of 411,779 nodes and 10,357,778 edges, using 27.3GB

of memory, that was solved in 15.2 hours, with the OOM killer killing the process during freeing memory in another 2.5 hours from there at over 33GB. The coarsening approach found the same maximum throughput as the MAS heuristic on the whole graph.

2.3. Random and Random Geometric Graphs

Naturally occurring graphs, are challenging to find in similar sizes to evaluate the robustness of algorithms to edge cases. As a result, many in researchers have turned to synthetic graphs. Generating synthetic graphs that exhibit similar characteristics to naturally occurring ones. Early in the field of graph theory, Erdös and Renyi investigated the behavior of random graphs [45].

In more recent research, some authors [32,39,89] have studied random geometric graphs. These graphs provide insight into wireless sensor networks and some naturally occuring phenomena. However, to understand social networks through synthetic graphs, a different class of graph is required. For the synthetic social networks, we use Random Typing Graphs, introduced by Akoglu in [3] with certain compelling properties discussed in more detail below.

We also create several random bipartite graphs to help illustrate the D_3 bound heuristic introduced later in Chapter 6. These graphs are not naturally occuring, except in rare cases. However, they allow analysis of the heuristic before attempting to extend it to other classes of graphs, whether random or not.

It can be shown that every MCFP instance has a set of critical edges that are saturated with flow by every optimal solution [93]. If the cut formed by removing the critical edges partitions V into two subsets, then the critical edges can be identified through the LP solution via duality theory.

2.3.1. Random Geometric Graphs

We generated 800 Random Geometric Graphs (RGGs) of 200 nodes each for evaluation. The nodes of an RGG are scattered randomly on a surface, in this case a unit square or a unit radius sphere; for each pair of nodes i and j, edge (i, j) is added to the graph if the

distance between i and j does not exceed a predetermined threshold. The threshold (r) for the unit square was varied from 0.12 to 0.39 using Euclidean ($L2$) distance. On the sphere, the threshold (θ) was varied from 0.4 to 0.96 radians. The four threshold values for each target were chosen to give approximately the same average degree 8, 12, 20, or 42 as shown in Table 2.1 below with their associated standard deviation σ values. In each case, 100 RGGs were generated to avoid sampling bias. We required that all graphs were connected. After uniformly randomly placing the points and connecting the nodes within the connection threshold, if the graph was not connected, the generation process was repeated by removing all nodes and starting over on that particular undirected graph.

Table 2.1. Thresholds and Degrees for the 200 Node RGGs

Threshold on Square	σ_{square}	Average Degree	σ_{sphere}	Threshold on Sphere
0.12	8.68177	8	8.31665	0.4
0.15	13.0619	12	12.67	0.5
0.20	22.0675	20	23.7641	0.7
0.30	44.4795	40	42.8654	0.96

In order to ensure that the randomly placed nodes were on the surface of a unit radius sphere, the (x, y, z) Cartesian coordinates were each uniformly chosen on the interval $[-1, 1]$. Then the magnitude of the position vector was calculated using the L^2 (Euclidean) norm. Any position vectors with a magnitude > 1 had their coordinates replaced with new random numbers until the magnitude was on the interval $[0, 1]$. This prevents biases toward poles. From there, the vectors were normalized to unit length, ensuring that they would be on the surface of the sphere. This approach is similar to the one used in [32] to generate RGGs on the unit radius sphere for their analysis.

The surface of the unit sphere is interesting because it does not have boundaries on the surface. In contrast to the unit square, where the nodes near the boundary will tend to have lower degree. The standard deviation of the node degree is in general slightly lower in the cases on the unit sphere, as shown in Table 2.1 above. In addition to the lower standard

deviation, the lower degree nodes are more evenly distributed within RGGs on the unit sphere than on the unit square.

2.4. Random Bipartite Graphs

Some of the approaches discussed later, such as the D3 bound heuristic in Chapter 6 for the shortest path bound of the MCFP, lend themselves to bipartite graphs. We created 6,210 random bipartite graphs for analysis. The generation algorithm creates a $G_{a,b,p}$ graph, where a and b are the size of each side of the partition and p is the probability of an edge being created between arbitrary nodes. Therefore, $G_{a,b,1.0}$ is equivalent to $K_{a,b}$. Some of the approaches discussed later refer to a "bipartite density", which we define as the proportion of edges in $G_{a,b,p}$ compared to the complete bipartite graph $K_{a,b}$, similar to the definition of graph density in the non-bipartite case. The random bipartite graphs were created with equal sized sets a and b from 2 to 70 and input probability p of existence of an edge from 0.1 to 0.9. For each edge between a and b, a random draw was made to create the edge. We created 10 graphs for each combination of set size and probability.

2.5. Random Typing Graphs

The concept of a Random Typing Graph (RTG) was developed in [3] as a simple method of creating synthetic graphs with community structure for social network analysis. In 1999, [4] and [13] developed the concept of "scale-free" graphs that obey the power law for node degree. That is, the distribution of the density of the nodes is similar to the distribution, first introduced by Zipf in [128] for the relative frequency of vocaulary word usage. These networks were analyzed for their robustness to node removal in [5] as measured by various properties. However, it was found in later work in 2009 that the degree power law was insufficient to describe social networks and that additional laws were needed, some being the power law applied to different properties. RTGs create graphs that obey all nine of those laws. The RTG uses a number of unique keys on a keyboard, and a space bar. Given a probability distribution of striking a key, all keys struck between spaces create a (possibly

empty string) node name. Each pair of node names generated form an edge. Using those names to uniquely identify nodes, nodes are added if a new unique name is created and edges have their strength increased if the edge already exists or are created if new. The process continues until certain conditions are met, such as the number of non-unique node names typed. Aside from graph generation, a similar technique using of pairs of words to represent edges and nodes has been used in sentiment analyis and authorship attribution [64].

In 2018, [25] found that based on the over 4000 naturally occurring networks stored at [36], scale-free networks were less common than previously believed. Results from that study and later ones have not so far yielded graph generators that build representative graphs. We assume for now that the additional laws satisfied by RTGs are close enough to these naturally occuring non-scale free (but close) networks that they represent a class of synthetic social network graphs sufficient for our analysis of MCFP heuristics in that class of graph.

The list of eleven laws that RTGs follow from [3] is summarized below for easy reference:

1. Power-law degree distribution - The number of nodes with a given degree should reduce according to a power law as the degree increases

2. Densification Power Law (DPL) - During graph generation, the number of edges should grow as $|E| \propto |V|^{\gamma}, \gamma > 1$

3. Weight Power Law (WPL) - The total weight of the edges compared to the number of edges should grow as $W \propto |E|^{\beta}, \beta > 1$

4. Snapshot Power Law (SPL) - Similar to the WPL, but applied to each node, the weight of edges attached to the node and the degree of the node should follow the power law

5. Triangle Power Law (TPL) - The number of triangles and the number of nodes that participate in those triangles should obey a power law, with an exponent < 0

6. Eigenvalue Power Law (EPL) - The eigenvalues of the adjacency matrix should follow the power law.

7. Principal Eigenvalue Power Law (λ_1 PL) - The largest eigenvalue λ_1 of the adjacency matrix and the number of edges should be power law proportional, with exponent $<$ 0.5

8. small and shrinking diameter - The diameter of the graph should be small and have a trend of shrinking with successive iteration of the generation process

9. constant size secondary and tertiary connected components - The smaller components tend to remain approximately constant size as the giant component continues to grow

10. community structure - The graph should have a modular structure

11. bursty/self-similar edge/weight additions - Edge additions to the graph should be bursty instead of uniform with spikes

The community structure is measured in terms of modularity, which has an inverse relationship to the beta parameter in the graph generator. Modularity has been traditionally defined in [101] and later in [108] as $Q = \sum_i (a_{ii} - b_i^2)$. Where given a set of communities, the fraction of all edges starting in community i and terminating in community j is denoted as a_{ij}. So, the fraction of intracommunity for community i is denoted as a_{ii}. This leaves $b_i = \sum_j a_{ij}$ as the fraction of all intercommunity edges originating in community i.

For our analysis, we generated 100 RTGs using two non-space keys, 0.5 probability of a space, 20000 words, and an off-diagonal bias (beta) of 0.2 to increase the modularity of the graph. According to the work from Akoglu, this bias provides an average modularity (community structure factor) of 0.6. These RTGs serve as synthetic graphs to evaluate at least the first cut in SNA style graphs. After creation, only the giant component of each RTG was retained, to ensure connectivity so that the MCFP would have non-trivial solutions. This process resulted in graphs with an average size of 140 nodes and 346 edges. These parameters were selected to give graphs with a max degree of about 20, so that they would be in line with one of the categories of RGGs used. The RTGs generated for study here had a diameter that ranged from 6 to 14 with an average diameter of 9.8 ($\sigma = 1.486$). The diameter of the RTG did not seem to have a relation to the error in an of our approaches. Some of these

graphs were large enough to stress the LP formulations, but were still evaluated using the heuristics described later in Chapter 4 and Chapter 5.

Chapter 3

LINEAR PROGRAMMING APPROACHES

Traditionally, the MCFP has been formulated as a linear program (LP). This requires a large amount of space, and limits approaches to serial ones. To get a full hierarchical result, the time required grows as $O(|V|)$ on top of the LP solver time. There have been a few advances in solving LPs using parallel techniques, but these have primarily been in the area of checking multiple hypotheses for selecting the basis pivot element, especially when some elements tie, and in the area of fast matrix manipulation.

In this chapter, we discuss the LP formulations and their potential extension to the hierarchical form of the MCFP. We then discuss some performance options relating to solver selection, using GPUs to improve runtime, and warm starting the LP with heuristic solutions.

3.1. LP Formulation

Matula and Shahrokhi formalized the definition of the MCFP in [112] as a linear program. They give both an edge-path and a node-edge formulation.

Presume a network of edges and nodes. On that network, edges have positive capacities, and there are non-negative demands between all pairs of nodes. The throughput represents the degree to which all demands can be satisfied by the network.

Any edges where the maximum concurrent flow is limited by the capacity are called "critical edges". That is, in all the LP formulations given below, the critical edges can be identified using the dual values of the capacity constraints at their upper bound. Only the edges with non-zero dual valued capacity constraints are considered critical, while other edges might be saturated but not constraining due to the nature of LP solvers. It can also be seen from all the formulations that the increase in throughput is a linear function of the increase in capacity of critical edges. As the capacity constraints are increased on critical

edges, additional edges will become critical and will need to be included in the set of edges getting increased capacity to keep this linear relation. The removal of critical edges generates a partition of the graph into at least two parts, which can be used for divisive average linkage clustering [94].

An instance of the MCFP is defined on an undirected graph $G = (V, E)$ where each edge $e \in E$ has capacity for c_e units of flow and there is demand for d_{ij} units of flow between each distinct pair of nodes i and j in V. Assuming, without loss of generality, that the nodes are numbered $1, 2, ..., |V|$, the set of distinct pairs of nodes in V is denoted by the set of ordered pairs $W = \{(i, j) : i \in V, j \in V, i < j\}$. Let P_{ij} denote the set of all paths with endpoints i and j, and let P denote the union of all sets P_{ij} for all node pairs $i, j \in V$ where $i < j$. The set of all paths that use edge $e \in E$ is denoted by P_e. The flow on a path $p \in P$ is represented by the decision variable f_p. The throughput is represented by the decision variable z. Using this notation, the MCFP may be stated by the following linear program (LP).

$$\text{maximize } z \text{ subject to :} \tag{3.1}$$

$$\sum_{p \in P_{ij}} f_p = z d_{ij} \qquad \forall (i, j) \in W \tag{3.2}$$

$$\sum_{p \in P_e} f_p \leq c_e \qquad \forall e \in E \tag{3.3}$$

$$f_p \geq 0 \qquad \forall p \in P \tag{3.4}$$

The objective function (3.1) maximizes the throughput for the concurrent flow. Constraint set (3.2) ensures that the same proportion of demand is met for all demand pairs. The combined flows of all paths using a particular edge e are limited to the edge capacity by constraint set (3.3), and the individual path flows are limited to non-negative values by constraint set (3.4).

The edge-path LP has $\binom{n}{2} + m = O(n^2)$ constraints and, since it must explicitly represent every path in the graph, could have more than 2^n variables though it was shown in [112] that there are only two active paths between any demand pair. Thus, the size of the edge-

path formulation grows exponentially as a function of the size of the input graph G making a polynomial-time implementation impractical for larger problem instances. Since a basic solution puts flow on an average of two paths per node pair, column-generation offers the possiblity of solving the edge-path formulation with relatively modest memory requirements. However this approach was found in [41] to be much slower than the approaches discussed later in this section.

The full edge-path formulation of the MCUP is as follows:

$$\text{minimize } u \text{ subject to :} \tag{3.5}$$

$$\sum_{p \in P_{ij}} f_p = d_{ij} \qquad \forall (i,j) \in W \tag{3.6}$$

$$\sum_{p \in P_e} f_p \leq u c_e \qquad \forall e \in E \tag{3.7}$$

$$f_p \geq 0 \qquad \forall p \in P \tag{3.8}$$

Given the above, if f is a solution with utilization u^* then $\hat{f}$ defined by $\hat{f}(p) = \frac{f(p)}{u^*} \quad \forall p \in P$ is a maximum concurrent flow of throughput $\hat{Z} = 1/u^*$.

The equivalent node-edge MCFP formulation given in [112] is a multi commodity flow formulation where each node (vertex) designates a distinct commodity for bookkeeping purposes, subject to a total capacity constraint on each edge, yielding a polynomially bounded linear programming formulation. The artificial distinction of separate commodity flows is unnecessary for many applications – in particular, identifying sparse cuts in a graph. The

node-edge formulation is given below as:

$$\text{maximize } z \text{ subject to :} \tag{3.9}$$

$$\sum_{k \in N(l)} (f_{lk}^i - f_{kl}^i) = \begin{cases} z \sum_j d_{ij} & \text{if } l = i, \\ -z d_{il} & \text{otherwise} \end{cases} \qquad \forall \text{ distinct } i, l \in W, \tag{3.10}$$

$$\sum_i (f_{lk}^i + f_{kl}^i) \leq c_e \qquad \forall \text{ edges } e = (l, k) \in E \tag{3.11}$$

$$f_{kl}^i \geq 0 \tag{3.12}$$

where f_{kl}^i is the flow of commodity i on the edge kl from k to l for all $i, k, l \in V$ and N (l) is the set of neighbors to l for all $l \in V$.

The node-edge form still requires $O(|V|^3|E|)$ space, using a constraint for each edge in the graph (up to $|E| = \binom{|V|}{2}$) and several slack variables in the LP formulation.

Recently, in 2015, [41] introduced a compact formulation for the MCFP based on node triples which yields LPs that are significantly smaller than those derived from either the node-edge or edge-path formulations traditionally used in the literature. For many applications, especially for SNA community detection, the size difference in the formulations becomes more prominent as $|V|$ increases. Computational results in comparing the solution times using the three formulations employing CPLEX (a state-of-the-art LP solver) on a set of problem instances from the literature and a set of instances defined on random geometric graphs indicate that the triples formulation can be solved faster than the other two formulations. This improvement is especially pronounced for problem instances defined on dense graphs. The triples formulation eliminates the dependency on the number of edges, resulting in a more consistently sized LP. There is still a $O(|V|^5)$ asymptotic space requirement, but smaller than other LP formulations.

The reduced triples formulation of the MCFP characterizes the flow between nodes i and j as either being direct, i.e., sent on edge (i, j), or diverted through a neighbor of $k \neq j$. If there is no edge between i and j, then all flow between the two nodes must be diverted. The

decision variable f_{ij}^k in the triples formulation represents the amount of flow between i and j that is diverted through k; that is, the total flow on paths composed of edge (i, j) followed by a path from k to j. The idea is to represent flow assigned to a (non-trivial) path between i and j as flow assigned to edge (i, k) adjoined with flow assigned to a path between k and j.

While the triples formulation affords significant memory savings compared to the node-edge formulation, it still requires over 1GB to solve relatively small graphs of 200 nodes and 2200 edges. This also takes several minutes on modern commodity hardware.

The triples formulation is given as:

$$\text{maximize } z \text{ subject to :} \tag{3.13}$$

$$zd_{ij} + \sum_{k \in V \setminus (i,j)} (f_{kj}^i + f_{ki}^j) - \sum_{k \in V \setminus (i,j)} f_{ij}^k \leq c_{ij} \qquad \forall i, j \in W \tag{3.14}$$

$$c_{ij} = \begin{cases} c_e & \forall \text{ edges } e = (i, j) \in E, \\ 0 & otherwise \end{cases} \tag{3.15}$$

$$f_{ij}^k \geq 0 \tag{3.16}$$

where the f_{ij}^k variables are constrained to be non-negative and defined on the set of node triples T given by $T = (i, j, k) : i \in 1, 2, ..., |V| - 1, j \in i + 1, i + 2, ..., |V|, i, k \in E$. The number of variables in the reduced triples formulation is at most $2|E||V|$, and the formulation has $\binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = O(|V|^2)$ constraints: one per distinct node pair. This gives a total asymptotic space usage of $O(|V|^3|E|)$. This is the formulation used in our implementations for evaluating the performance of our heuristics.

3.2. Hierarchical LP Formulation

In [90], Mann started the extension of the MCFP to the hierarchical form involved fixing a known Z_i for each cut, and maximizing Z_{i+1} on the remaining demand pairs until all are separated when critical edges are removed. This was used to detect community structure in

social networks. That is, finding which edges are in dense subgraphs and which are between dense subgraphs. However, the implementation used in that paper was a divisive model, rather than a hierarchical formulation. Here, we refine the definition of the hierarchical formulation and use it for clustering, along with a discussion on the difference in results between the two formulations.

The MCFP can be extended to the hierarchical case, with an approach that successively solves the LP by reformulating certain constraints after each iteration. That is, beginning with the non-hierarchical formulation, solve for z. Then that z value is used in the demand constraints for any node pais separated that would be separated by the removal of critical edges. Now, the LP is solved again for a new z value, to be added to the newly found critical edges for which the z was not previously known. This process is repeated until a z value is found for every node pair in the graph.

For each iteration, z has been shown above to be a piecewise linear function of the capacities of the edges cut. It may be possible to manipulate the capacities of the critical edges instead of fixing the z values when solving the next iteration of the hierarchy. If that is possible, then other existing algorithms for solving the MCFP may be extended to the hierarchical case, without going to a strictly divisive case.

One particular challenge to the method of manipulating the capacities is to determine what the new capacities should be. This is only further complicated in the non-uniform capacity case. If this can be learned from the results of the first iteration of the algorithm, then the search for the second iteration becomes easier, and likewise for each successive iteration. Another approach would be to attempt to use the results from the first iteration to determine the entire hierarchy.

We have extended the MCFP reduced triples formulation to the hierarchical case to look at the first few cuts in several RGGs. The results are similar to the node-edge formulation described in Mann's dissertation [90]. Like other algorithms discussed here, we compare the results of the algorithms results on unit square and unit sphere RGGs, half-random geometric graphs, and grid graphs.

3.3. Formalization of the HMCFP

Multiple levels of sparse cuts, approximated by maximum concurrent flow, were used for clustering purposes in [90]. In that paper, the maximum concurrent flow problem (MCFP) was solved iteratively on each cluster as each cut created new clusters. Since the MCFP requires significant quantities of memory to solve, finding solutions on each cluster is a heuristic for tractability.

In contrast to the discussion above, the hierarchical formulation of the MCFP (HMCFP) involves re-solving the problem for the whole graph at each step, similar to the approach taken in [29] and [30]. This is partly enabled by more capable computers since the 2008 publication, and partly by more efficient formulations of the problem, such as the triples formulation described in [41].

The original edge-path formulation of the problem was given above in (3.1), with the node-edge form shown in (3.9), and the triples formulation given in (3.13)

The hierarchical formulation extends the model using Z_{known}, the set of (i, j) pairs with a known z value between them. Then z_{ij} is the known z value for pair $(i, j) \in Z_{known}$. The maximum throughput at step s is represented as z_s. The resulting hierarchical edge-path form is:

$$\text{maximize } z_s \text{ subject to :} \tag{3.17}$$

$$\sum_{p \in P_{ij}} f_p = \begin{cases} z_{ij} d_{ij} & (i,j) \in Z_{known} \\ z_s d_{ij} & (i,j) \notin Z_{known} \end{cases} \qquad \forall (i,j) \in W \tag{3.18}$$

$$\sum_{p \in P_e} f_p \leq c_{ij} \qquad \forall e \in E \tag{3.19}$$

$$f_p \geq 0 \qquad \forall p \in P \tag{3.20}$$

The resulting hierarchical node-edge form is:

Maximize z_s subject to : (3.21)

$$\sum_{k \in N(l)} (f^i_{lk} - f^i_{kl}) = \begin{cases} z_{ij} \sum_j d_{ij} & \text{if } l = i \text{ and } (i,j) \in Z_{known}, \\ -z_{ij} d_{il} & \text{if } l \neq i \text{ and } (i,j) \in Z_{known}, \\ z_s \sum_j d_{ij} & \text{if } l = i \text{ and } (i,j) \notin Z_{known}, \\ -z_s d_{il} & \text{if } l \neq i \text{ and } (i,j) \notin Z_{known} \end{cases} \quad \forall \text{ distinct } i, l \in W, \tag{3.22}$$

$$\sum_i (f^i_{lk} + f^i_{kl}) \leq c_e \quad \forall \text{ edges } e = (l,k) \in E \tag{3.23}$$

$$f^i_{jk} \geq 0 \tag{3.24}$$

And the hierarchical triples formulation becomes:

Maximize z_s subject to : (3.25)

$$z_{ij} d_{ij} + \sum_{k \in V \setminus (i,j)} (f^i_{kj} + f^j_{ki}) - \sum_{k \in V \setminus (i,j)} f^k_{ij} \leq c_{ij} \forall i, j \in W \text{ and } \in Z_{known} \tag{3.26}$$

$$z_s d_{ij} + \sum_{k \in V \setminus (i,j)} (f^i_{kj} + f^j_{ki}) - \sum_{k \in V \setminus (i,j)} f^k_{ij} \leq c_{ij} \forall i, j \in W \text{ and } \notin Z_{known} \tag{3.27}$$

$$c_{ij} = \begin{cases} c_e & \forall \text{ edges } e = (i,j) \in E, \\ 0 & otherwise \end{cases} \tag{3.28}$$

$$f^k_{ij} \geq 0 \tag{3.29}$$

3.4. Derivation of the Hierarchical Sparsest Cut

Since the Sparsest Cut Problem is so closely related to the Maximum Concurrent Flow Problem, and our MAS heuristic uses sparse cuts, we define here the Hierarchical Sparsest

Cut Problem for later use. The Sparsest Cut Problem seeks to minimize the ratio of cut capacity to separated demand. The implementation of this in the MAS heuristic is discussed later in Chapter 4 and the extension of the ratio to the hierarchical case is also discussed in detail. The mixed integer linear programming formulation of the SCP was given in [57], derived from the dual of the edge-path formulation of the MCFP given in 3.1. This differs from the MCUP given in 3.5 in that it does not attempt to map the resulting dual variables on to the intuitive concepts used to formulate both the MCFP and MCUP.

The extension to the hierarchical form using the same concepts as the MCFP is not simple. A similar approach to the extensions above for the MCFP could be used and then similarly take the dual at each step in the hierarchy while reintroducing the binary variables. For our purposes, especially in Chapter 4, it is sufficient to define the concept of residual capacity in the MCFP so that the heuristic can be made hierarchical. Consider that some amount of capacity on each slack edge may be used to send flow from a node not adjacent to a critical edge across the cut formed by the critical edges. In order to evaluate the amount of flow that might be available for the next cut while maintaining a hierarchical rather than divisive case, this flow must be removed from the calculation for the next cut. The residual capacity on that edge is then the amount remaining. The equations and a more detailed walkthrough of an example are available in Chapter 4.

3.5. Performance of LP Solving Methodologies

LP solvers solve the input problem by either the Simplex algorithm or interior point methods. These methods have been covered extensively in literature, so only a brief review is given here for convenience. Both methods construct a convex polytope from the constraint matrix, where any optimal solution is guaranteed to be at a vertex of this shape. This does not preclude multiple optimal solutions at different exterior points.

The Simplex method begins by picking an initial basis where several of the variables are set to zero, so that a feasible solution can be found with the remaining variables. This step is an active field of research, since it can take significant time to find [20, 71]. In our

implementations using GLPK, getting to a nontrivial basis sometimes took half or more of the total solver time. From there, the algorithm proceeds from vertex to vertex, along the exterior of the convex polytope. Each step is to a neighboring vertex with the most optimal solution to the objective function out of all the neighbors, with ties broken arbitrarily. The algorithm ends when there are no more neighboring vertices that can improve the solution. In order to take each step, a pivot row and column are identified, and all other rows are updated by subtracting the product of the current row's pivot column and the new pivot row. Given that, most of the computational time spent in the algorithm is in basic row updates. Further, there are an exponential number of exterior vertices, which could result in a long runtime for large problems like the MCFP.

In the 1980s, interior point methods appeared as a polynomially bounded approach to solving linear programs. These approaches only reach an exterior boundary when they find an optimal solution. They begin with an initial point within the convex polytope of constraints. The objective function is also modified to have a barrier function to keep the incremental progress from touching a boundary prematurely. From there, the process is similar to gradient descent along the objective function gradient. This is modified to include operating in a transformed space with additional dimensions and calculating the projection into the lower dimension convex polytope at each step. This modification means that most of the computational time is spent in matrix operations like multiplication, inversion, and transposition. However, decades of optimization in matrix libraries [10, 59] and hardware to execute them, means that interior point methods should be more ideal for acceleration on coprocessors like GPUs, FPGAs, or some of the emerging machine learning ASICs [110], like the Google TPU, AWS Inferentia, or Intel NNP-I/T chips among others.

In solving the MCFP for the NCAA football data with GLPK, the solution is found in roughly 15 minutes on a recent laptop using the Simplex algorithm. When using interior point methods, it took close to 2 hours. However, in tests with CPLEX, the interior point methods showed significant benefit over Simplex methods. In [41], the authors found that the performance the LP algorithms solving the node-arc and triples formulations was

significantly affected by the size and density of the graph, and the number of non-zero d_{ij} values independent of hardware. In the studies presented in later chapters, we noticed that a higher number of non-zero d_{ij} values created a constraint matrix with a higher number of non-sparse columns, causing GLPK to change the matrix algorithms used to less efficient forms. We also noticed that non-uniform capacity caused both the interior point and Simplex methods to run more slowly with around 100 nodes than uniform capacity cases of 200-300 nodes. While certain problems may be better suited to one method or the other, the specific implementation in a solver library must be considered when selecting which one to use for a certain problem. In this case, GLPK has focused for years on the Simplex algorithm and only provides minimal support for interior point methods. In contrast, CPLEX has focused on interior point methods, which can be better scalable through GPU or FPGA style vectorization.

3.5.1. GPU Computing

Recent advances in general purpose GPU (GPGPU) computing have included vendor-specific and common languages to utilize the GPUs many floating point units, often numbering in the several hundreds or thousands, for parallel computation. Early work on GPU acceleration of LP was primarily focused on interior point methods that are heavily dependent on matrix inversion or decomposition. Other approaches have implemented Cholesky decomposition on the GPU to more quickly solve LPs using interior point methods [40, 70, 85, 124]. The matrix inversion step is another area that benefits from GPU acceleration, explored in [69] and [115]. All of these approaches gained in general at least a 10x speedup.

It was not until years later that researchers were able to successfully accelerate Simplex algorithm implementations [60, 78, 106, 107]. These approaches in general had less than a 10x speedup, but still showed benefit for some problems. An in-depth review of prior art in this area up to 2018 can be found in [53]. However in 2019, an implementation of the Simplex algorithm in [60] showed over a 10x speedup once the problem size grew to over 200 dimensions in the LP, as compared to both GLPK and CPLEX instances on the CPU.

There are also several implementations in CUDA and OpenCL of the Floyd-Warshall all-pairs shortest paths algorithm available on gitlab. While these do not directly apply to the LP formulations, they may be useful in accelerating some rerouting approaches.

3.6. Conclusion

We have formalized the description of the Hierarchical Maximum Concurrent Flow Problem in contrast to the divisive version presented previously. While both are based on the duality of the MCFP and the Sparsest Cut Problem, the divisive approach using subgraphs ignores interactions that are captured in the new hierarchical formulation. Equations for the edge-path, node-edge, and the triples formulation are shown for the first cut and hierarchical forms. We also covered the concept of residual flow for use in hierarchical heuristics. Finally, this chapter reviews the different LP solving methodologies and begins to discuss the concept of speedup through vectorization of operations on GPUs.

Chapter 4

APPROXIMATING SPARSEST CUT WITH MAXIMUM ADJACENCY SEARCH

The Hierarchical Maximum Concurrent Flow Problem has shown significant utility in finding community structure in graphs. However, the time and space complexities of the formulation limit feasibility to relatively small graphs. Some of our previous studies have shown that using the Maximum Adjacency Search can find the Maximum Concurrent Flow in many cases with lower time and space requirements. We formalize that heuristic, extend it to the hierarchical case, and show how it can be used to calculate the approximate flows within the graph. We provided an initial discussion on this topic in [119] and extend the discussion in this chapter.

4.1. Maximum Adjacency Search

The maximum adjacency search (MAS) was introduced in [95] as a method of approximating edge connectivity within a graph. The MAS is initialized by setting the reach-count for each node to 0 and adding the set of all nodes to a keyed updatable priority queue where the key is the reach-count. Then, beginning with the designated start node, all non-visited neighbors of the current node have their reach-count incremented. The current node is marked as visited and the next node is selected based on the highest reach-count in the queue, with ties broken arbitrarily. The process is repeated until all nodes have been visited.

The Maximum Adjacency Search was introduced in [95] and used to partition graphs in [28]. It was also modified to find the minimum cut in [117]. This Stoer-Wagner min-cut approach was added to the Boost C++ library in version 1.45.0 and later modified by us to use a visitor pattern and expose the underlying maximum adjacency search algorithm in version 1.53.0 The nature of the search algorithm tends to visit nodes within a given community before expanding to a new one. This coincides with the tendency of sparse cuts

to not split communities in a graph.

The MAS was later used in [117] to find the min-cut of a graph in $O(|V||E|+|V|^2 \log |V|)$ time. The search was repeated starting at each node to find the min-cut, much like network flow approaches that check the results for each possible sink node in the graph. However, where the network flow approaches, like the one from Goldberg and Tarjan [56] operate in $O(|V||E| \log(|V|^2/|E|))$ time, the MAS approach is faster.

The MAS was also used for graph partitioning in [28] by calculating the number of edge disjoint paths between pairs of vertices. The partitions were then established by analysis of breakpoints where the number of pairwise edge disjoint paths decreased before increasing again.

The Stoer-Wagner min-cut algorithm was added to the Boost Graph Library (BGL) [113] in version 1.45.0, with the MAS step hidden as an implementation detail. We modified the implementation to expose the MAS and submitted a patch to the BGL maintainers including documentation of the algorithm. The patch was accepted for version 1.53.0 of the library. Our implementation contains a visitor concept which we used to create a min-cut visitor for the Stoer-Wagner algorithm which was also submitted as part of the patch.

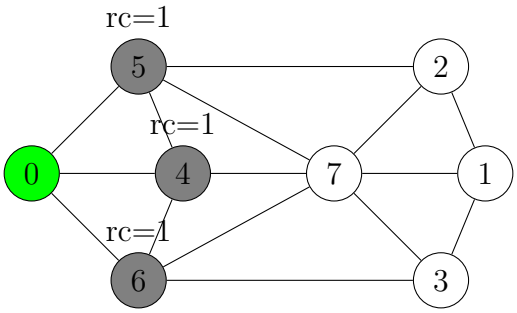
4.2. Use as a Heuristic

In research on the MCFP, we have also created an MAS visitor that tracks the cut density in an effort to obtain a heuristic for the sparsest cut.

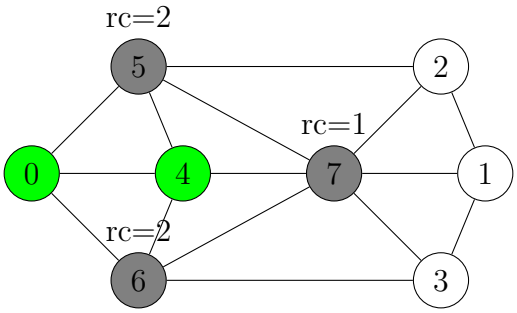
The sparsest cut visitor that evaluates the cut density at each node, and keeps track of the best solution so far. We execute the $O(|E|)$ search multiple times, starting from each node in the graph to expand the solution space of cuts evaluated, for a total of $O(|V||E|)$ complexity. In the case of a new graph, the result of the heuristic will show nodes on either side of the partition, the edges cut, and the cut density. This is not as much information as is available from the linear programming formulations above, but requires significantly less computation and only $O(|V|+|E|)$ space compared to $O(|V|^3|E|)$ in the triples formulation. Where our previous work on LP approaches to the MCFP has been limited to graphs of a

few hundred nodes, we have used this heuristic to evaluate graphs of a few thousand nodes.

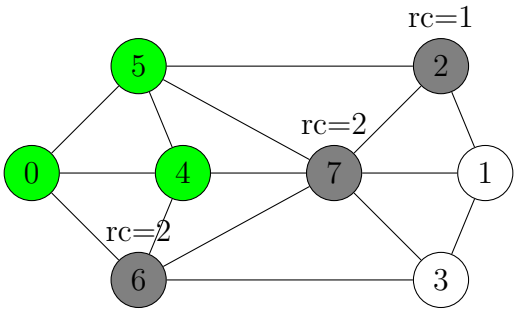
For a brief walkthrough in Figure 4.1 , we use the same G3 graph as in the other examples below for consistency. We start the algorithm at node 0, resulting in a cut of 3/7, which will improve to 5/16=0.3125 as the algorithm proceeds. At each step, any nodes that have been reached (gray) but not visited (green) have the reach count indicated above them. To break ties between the nodes of the highest reach count, we select the node least recently added to the keyed updatable priority queue. This walkthrough only shows the case for a single starting node, where in practice, the algorithm is repeated from each starting node. The choice of tie-breaker can drive the answer, which is somewhat mitigated by the repetition from each node, but the tie-breaker choice here does find the sparsest cut on this pass.



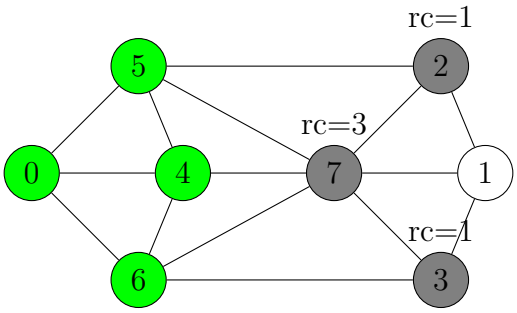
(a) node 0 visited (den=3/7)



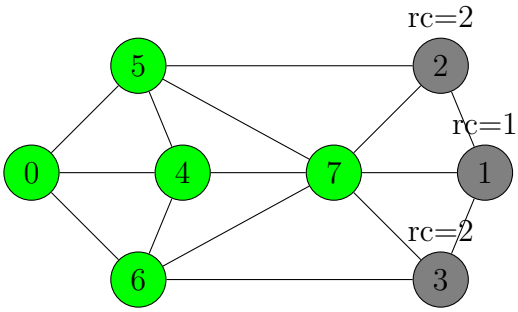
(b) node 4 visited (den=5/12)



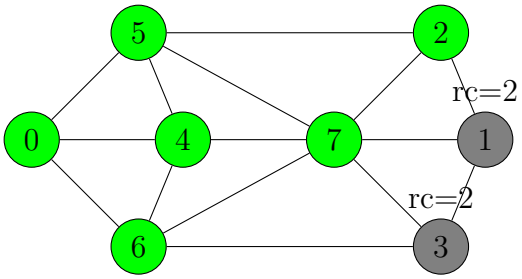
(c) node 5 visited (den=5/15)



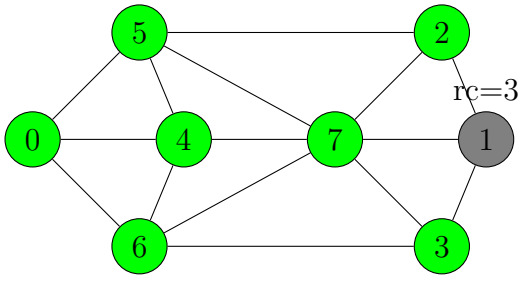
(d) node 6 visited (den=5/16)



(e) node 7 visited (den=5/15)



(f) node 2 visited (den=4/12)



(g) node 3 visited (den=3/7)

Figure 4.1. MAS walkthrough graph 3 (best sparsity = 5/16)

4.3. Hierarchical Cases

To extend to the hierarchical case, we introduce new terms to the definition of the Sparsest Cut. The sparsest cut is defined as the cut that minimizes cut density $d = \frac{\sum_{cutedges} capacity}{\sum pairsdemand}$. In the hierarchical case, we have to additionally consider the residual capacity and the previously met demand. Imagine a graph partitioned into $|A|$ and $|B|$ by the first cut, having throughput $z_1 = 1/d_1$. Let cut 2 separate subset $|A|$ into $|AA|$ and $|AB|$. To calculate the density of the second cut, d_2, we must determine how much of the capacity of superedge $|AA| - |AB|$ is dedicated to flow from $|AA|$ to $|B|$, given by the capacity of superedge $|AB| - |B|$ reduced by z_1. So the residual capacity can be calculated as $c_{residual(|AB|-|B|)} = \sum_{|AA|-|AB|} capacity - (z_1 * \sum_{edges} |AB| - |B|)$ Further, we only consider demand between $|AA|$ and $|AB|$. This results in $d_2 = \frac{\sum_{|AA|-|AB|} capacity - c_{residual(|AB|-|B|)}}{|AA|*|AB|}$.

A limitation of the MAS heuristic approach in the context of our research is that there has been no obvious way to extend it to the hierarchical case. More research is needed to see if the MAS approach can also approximate cuts beyond the first one. In many cases, for RGGs, the MCF solution only separates the graph into two parts, where Matula showed that for cuts into less than five parts, the MCF solution is the sparsest cut [98]. Therefore, we may be able to analyze the resulting components using residual capacity calculations while disallowing edges between the components to be cut and find a second cut.

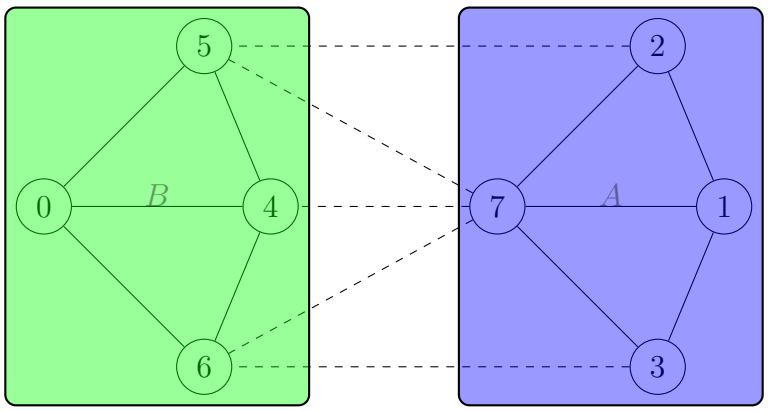


Figure 4.2. First Cut of graph 3 ($z_1 = 0.3125$; B=0,4,5,6; A=1,2,3,7)

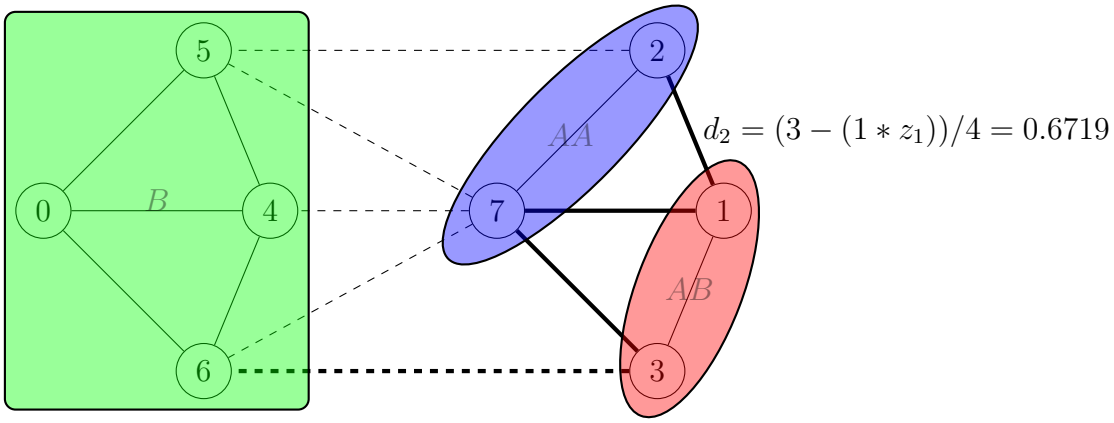


Figure 4.3. Second Cut of graph 3 ($z_2 = 0.340909$; AA=2,7; AB=1,3)

By repeating the calculation of this heuristic (or the exact sparsest cut), an upper bound on all flows in the node-edge formulation of the HMCFP can be obtained. By exploiting the dual relationship of the MCFP and the Sparsest Cut Problem, it may be possible to warm start an LP. Future work in this field includes an exploration of the possibility of the warm start, and a transformation of the flows to the triples formulation for more efficient calculation.

4.4. Approach

As described in Chapter 2, we use over 6000 random bipartite graphs, 500 random geometric graphs on the unit square, 400 random geometric graphs on the unit sphere, and 100 random typing graphs to compare the LP formulation of the MCFP against our heuristic. We compare the throughput values calculated as well as the time and memory utilization for the calculations. We also use some selected graphs from [19, 91] for specific illustrations.

It was shown in [112] that finding the sparsest cut is not guaranteed to be found by a polynomially bounded formulation. As long as the graph is separated into fewer than five parts, the LP finds the sparsest cut. The 5 partition case may also result in a $K_{3,2}$, which represents the smallest case of the polynomially bounded formulation failing to find the sparsest cut.

In the cases of five or more partitions at a given cut without ties, it may be argued that this represents sufficient homogeneity in the data to stop attempting to split the nodes from each other at that level. A five (or more) part case with ties or for cases with less than 5 partitions, the data still represent different clusters and the hierarchical partitioning should continue.

As an example, we use graph 3 from [19], with the first two cuts shown in Figure 4.4 and Figure 4.5. For cut 1, the calculation is simple: $d = \frac{5}{4*4}$. To illustrate the extension to a hierarchical case, the cut 2 is $d_2 = \frac{4-1}{2*6-3.2}$. This continues for cuts 3 and 4 (not shown) with $d_3 = \frac{2}{3*5-9.6}$ and $d_4 = \frac{1}{7*1-5.4}$, respectively. The heuristic also finds these values, indicating a continued utility as the hierarchy is created.

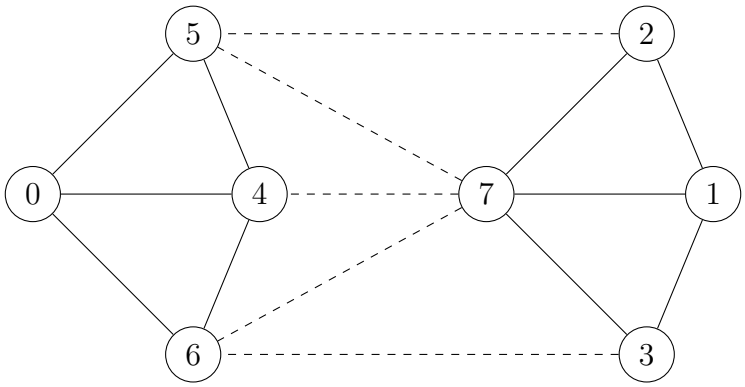


Figure 4.4. First Cut of graph 3 (density = 0.3125)

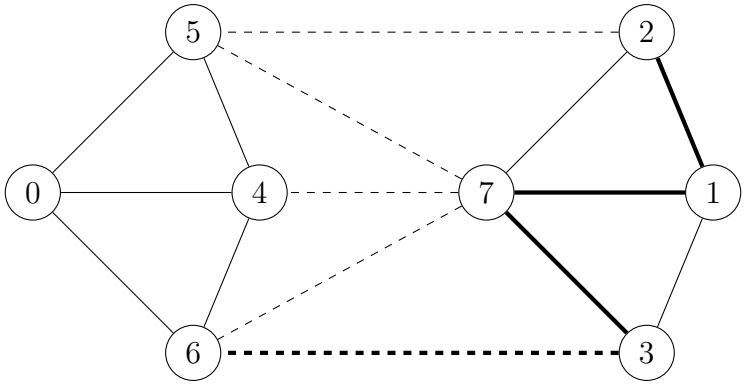


Figure 4.5. Second Cut of graph 3 (density = 0.340909)

4.5. Discussion

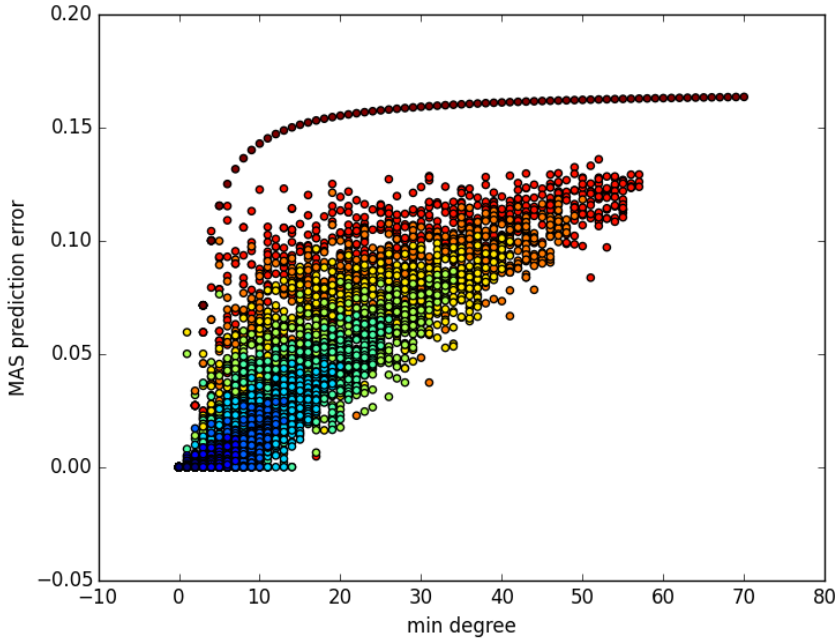
4.5.1. Relationship to MCFP

Any cut is an upper bound on the Sparsest Cut, so the heuristic is searching for progressively sparser cuts to approximate the sparsest one. The MCFP is also bounded from above by the SCP. Therefore, the MCFP solution must be less than or equal to the result of the MAS heuristic.

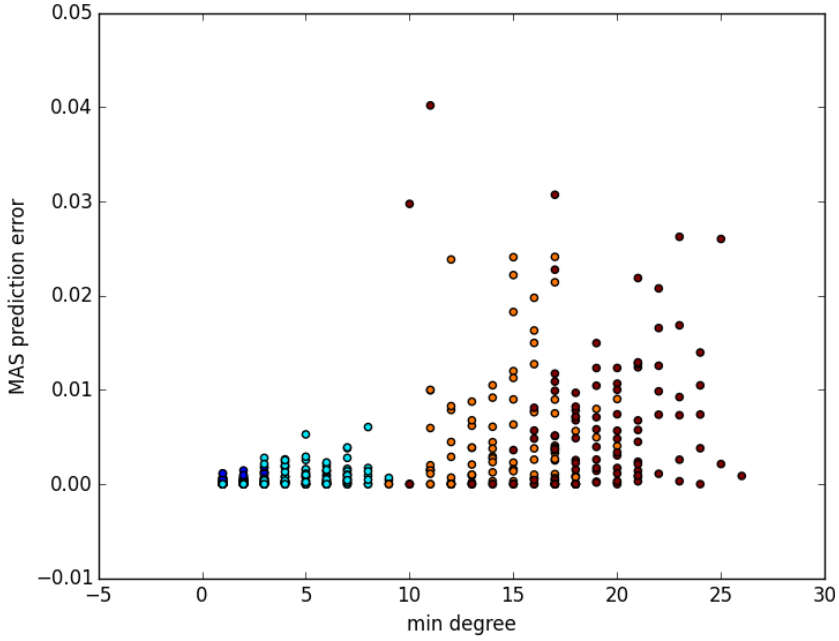
4.5.2. Performance Discussion

We can see in all examined cases that the total error is small (less than 0.10%) of the LP solution value. This error is measured by treating the LP z value as truth and examining the MAS value as a deviation from it, that is: $z_{MAS} - z_{LP}$, or for percentage: $\frac{z_{MAS} - z_{LP}}{z_{LP}}$.

The variation in the error also gets larger as the minimum degree of the graph grows, as shown in Figure 4.6. In the case of the random bipartite graphs, larger graphs are more likely to result in a "web cut" where the graph breaks into more than two partitions simultaneously. In the random geometric graph cases, the errors are more pronounced on the square than on the sphere, due to the boundaries of the field impacting the minimum degree and the overall graph structure. The random typing graphs all have the same minimum degree of one, since there are "leaf nodes" created in real and synthetic social network graphs. This makes analysis of performance vs min degree challenging in those cases. However, the correct solution was found in 86 out of the 100 RTG cases.

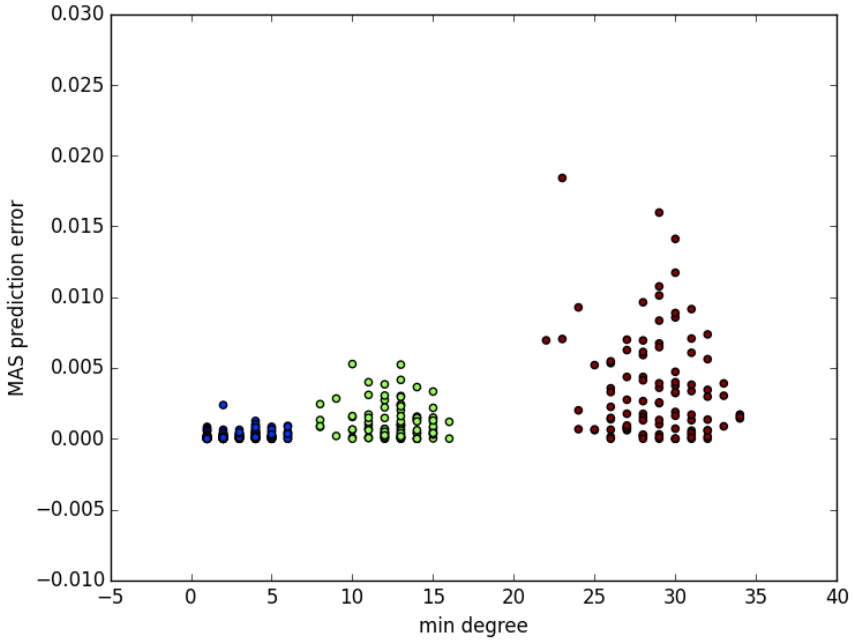


(a) random bipartite

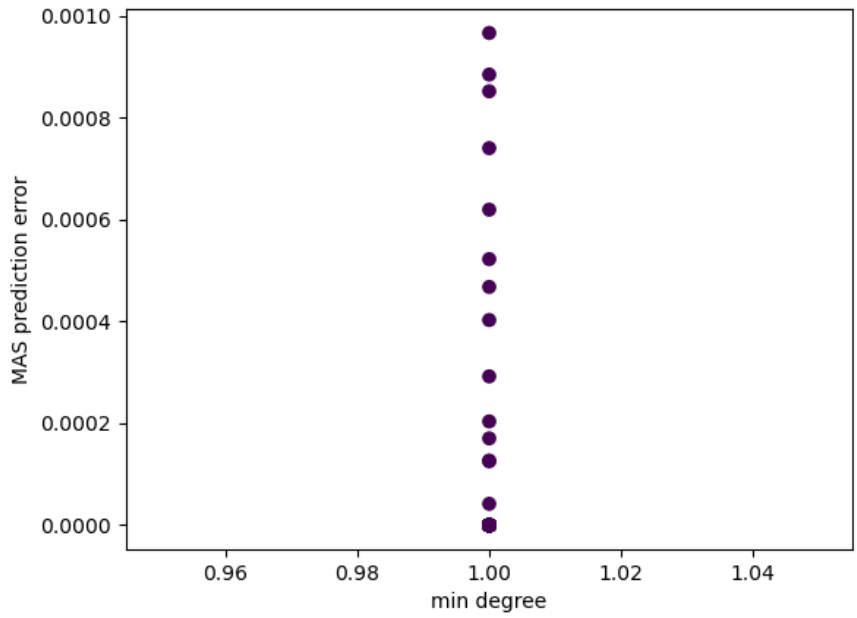


(b) RGG square

Figure 4.6. Comparison of MAS heuristic to the LP solution



(c) RGG sphere

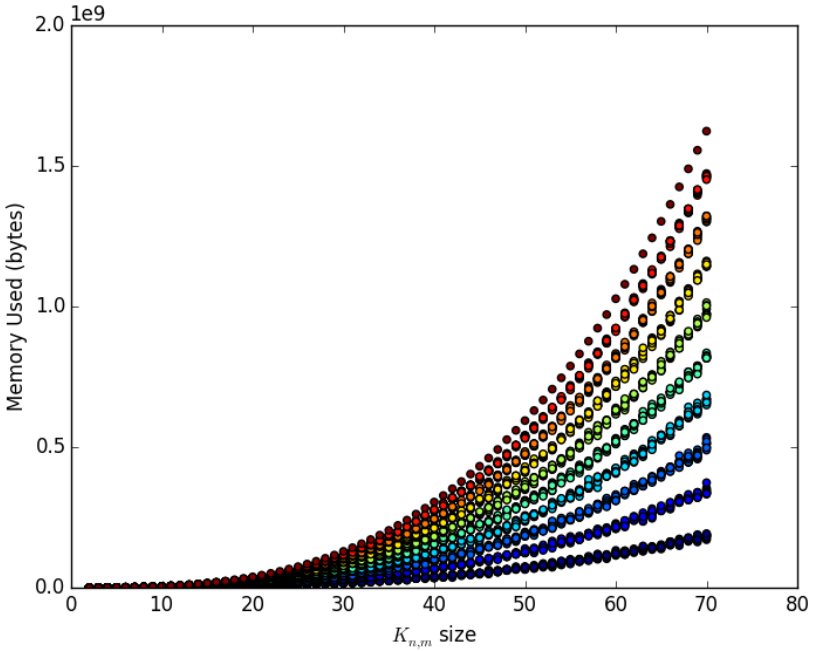


(d) RTG

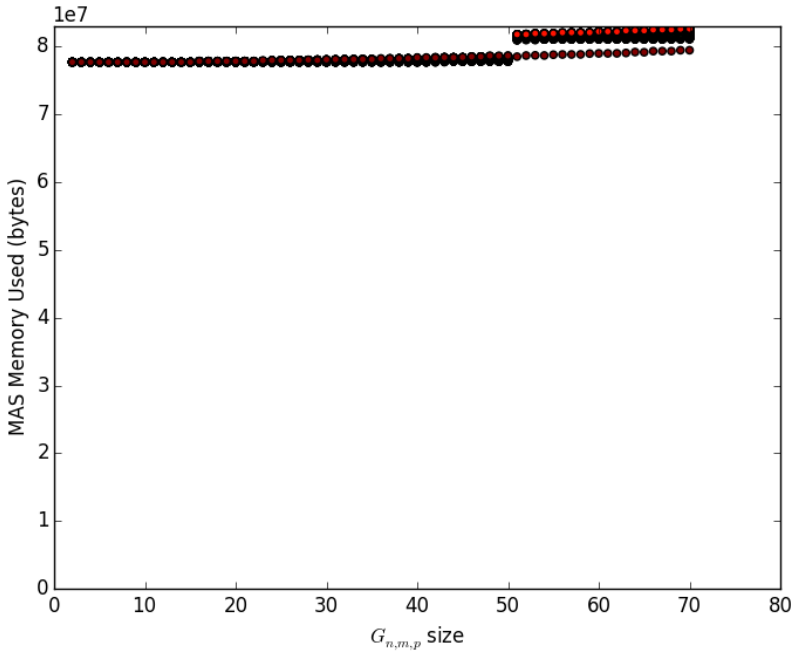
Figure 4.6. Comparison of MAS heuristic to the LP solution (cont.)

To obtain these results, the heuristic consistently used 7-8Mb of memory versus gigabytes. The RGG cases grew to almost 4Gb in the largest cases. In the random bipartite cases, the LP space utilization is easy to see in Figure 4.7.

The heuristic is also calculated in under 1 second for most these graphs compared to a few hours for the larger cases in the LP. These results are for the first cut only, where the differences are amplified when cuts deeper in the hierarchy are analyzed.



(e) LP (triples formulation)



(f) MAS heuristic

Figure 4.7. Comparison of memory used to solve a random bipartite graph

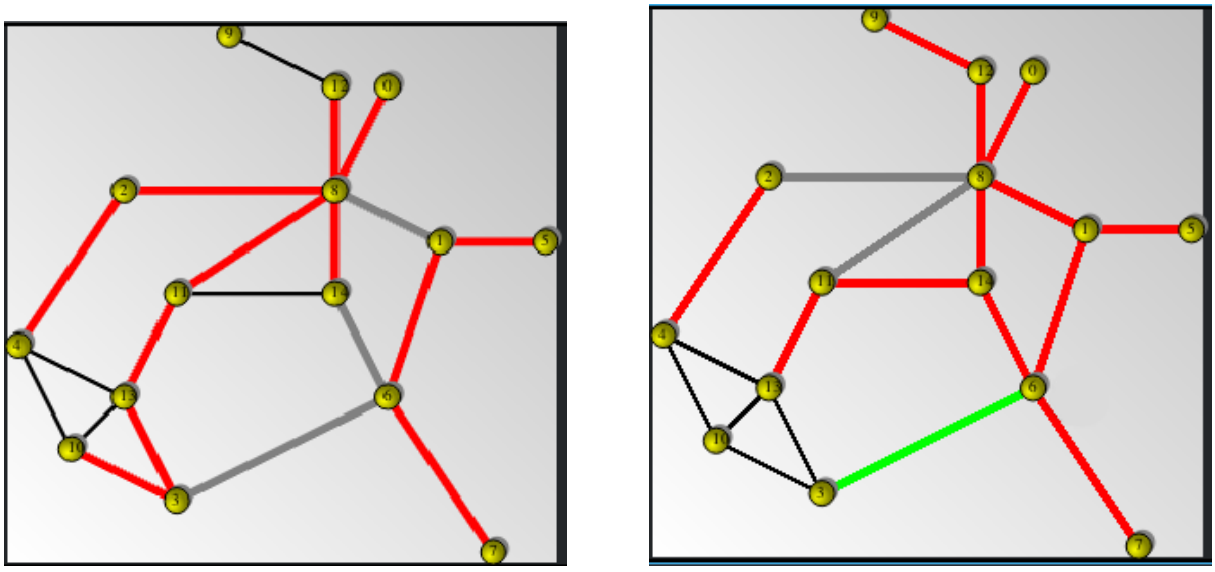
A common small graph to use in network analysis is the social network on the Florentine Families. Despite being a small graph, it has several interesting features, including breaking into a "web cut" later in the hierarchy. We compare the results of the hierarchical form of our heuristic to that of the triples formulation of the LP as an example of its utility. In Table 4.1, we see the partitioning given by the hierarchical triples formulation of the LP. In the hierarchy, the cut at 0.13697 is a web cut, separating the graph into several parts. The heuristic continues to separate the graph correctly up to this point. The fact that the heuristic gets this cut correct is somewhat surprising, though likely attributable to the depth in the hierarchy at which it is encountered. This is also an example of the nuances of a hierarchical approach, rather than a divisive approach, which would not have found the correct answer as shown in the non-hierarchical cases analyzed earlier in this paper.

In most of the hierarchical runs, the MAS approximated value is less than the LP value. While the values are different, the critical edges identified are similar, especially in the early portions of the hierarchy. By looking at the z values from the bottom up, a set of flows between each partition can be calculated. At the bottom, edges will contain z units of flow between the separated nodes. As the the hierarchy is traversed upwards, the flow into a node will be used as flow-through to reduce the total flow that may come out of a given node, so that all flow from one supernode to another is properly counted.

However, the next cut after the web cut (shown in bold) is not found correctly by the heuristic, as shown in Figure 4.8. The heuristic separates six nodes on the left from the rest of the graph by cutting the graph into two parts. In contrast, the LP solution cuts the graph into 3 parts, consisting of node 3, the right-most four nodes, and the rest of the graph. Since the heuristic does not account for web cuts, it cannot appropriately track the known z values resulting from them during subsequent cuts.

Table 4.1. Hierarchy of Florentine Families graph

Cut Value	MAS Value	LP Partition index by node														
0.0384615	0.0384615	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1
0.0653846	0.06	0	0	1	1	1	0	0	0	0	2	1	0	2	1	0
0.0851648	0.0714286	3	0	1	1	1	0	0	0	0	2	1	0	2	1	0
0.0851648	0.0714286	4	0	1	1	1	2	0	0	0	3	1	0	3	1	0
0.0851648	0.0714286	3	1	4	4	4	2	1	5	1	0	4	1	0	4	1
0.0888278	0.0769231	5	3	1	1	1	6	3	0	2	4	1	2	4	1	2
0.13697	0.09	8	0	3	2	2	4	9	7	5	6	2	1	6	2	1
0.262036	0.130435	5	9	7	8	4	2	0	1	10	6	4	3	6	4	3
0.417373	0.147541	11	3	10	2	9	8	4	6	0	1	7	5	1	7	5
0.5	0.214286	7	2	6	9	10	0	12	3	11	5	4	1	8	4	1
0.54911	0.6	9	7	5	6	8	2	1	11	4	10	13	3	0	13	12
0.630769	1.0	0	4	3	12	6	8	2	9	1	10	5	13	7	14	11



(a) LP (triples formulation) (b) MAS heuristic

Figure 4.8. Comparison of cut results deep in the hierarchy

4.6. Parallelization

For both the Stoer-Wagner min cut and the MAS with a sparse cut visitor, the algorithm is run with each node as a starting point. If we skip the assignment phase of the Stoer-Wagner algorithm, both approaches are easy to parallelize. Each iteration with a different starting node is independent from the others, so a threaded approach is obvious. A future parallel implementation study could include constructing a thread-safe data structure for the Stoer-Wagner assignment phase to keep the efficiencies that provides while still permitting a threaded approach.

4.7. Conclusion

In this chapter, we have reviewed the formulation of the Hierarchical Maximum Concurrent Flow problem and analyzed a heuristic using the Maximum Adjacency Search on multiple random instances of three different types of graphs. In the non-hierarchical cases, the heuristic found the correct answer in several graphs, and was particularly accurate on smaller cases that were unlikely to contain web cuts. Whether the error in the heuristic is acceptable or not, given the memory and speed improvements, will depend upon the particular use cases for a given application.

We further explored the utility in the hierarchical case. The LP formulations of some of the larger cases can take days, where the heuristic gets an approximate solution in seconds. In the case of the Florentine Families graph, the graph structure found is the same in both cases until fairly deep in the hierarchy. This small graph does not show the same memory and speed improvements, but is checkable by hand for the full.

4.8. Future Work

In an attempt to steer the MAS towards the sparsest cut more quickly, we initialized the each node with a reach-count of the negative of its degree instead of 0. Then the reach-count is incremented by 2 instead of the default of 1. Preliminary investigation showed that for some graphs, this yielded a sparser solution than the original implementation. However for

others, there was no effect. A full comparison of this method to the original implementation as a part of future work should investigate whether these changes to the algorithm can make the prediction worse in some cases. If it cannot, but in some cases provides a better answer, then it would be a better heuristic.

Chapter 5

GRAPH COARSENING FOR RUNTIME IMPROVEMENTS IN THE MAXIMUM CONCURRENT FLOW PROBLEM

The Maximum Concurrent Flow Problem (MCFP) has shown significant utility in finding community structure in graphs. However, the time and space complexities of the formulation limit feasibility to relatively small graphs. Some of our previous work has used a heuristic to estimate the solution in larger graphs. In this paper, we use a similar method to coarsen the graph to use the full linear programming approach to solve the MCFP. Most naturally occurring graphs, such as those in social networks, follow a power law distribution of degree. Therefore, a coarsening approach that tends to group graph communities creates, with high probability, super-nodes that will not be internally separated as part of the real solution of the MCFP. This paper explores the use of the Maximum Adjacency Search (MAS) as a coarsening algorithm as a community detection heuristic, extending the work we originally presented in [120].

We explore the use of sparse cuts for divisive average linkage clustering with a focus on the resulting interactions between the clusters. The sparse cuts are approximated using the Hierarchical Maximum Concurrent Flow Problem. The interactions between the clusters are represented as graphs of supernodes with superedge connections. The resulting possibilities of the first few divisions are presented. We also show that a robust solution without ties must be triangle free. An example is presented using a sample graph for clarity.

5.1. Introduction

Historically, hierarchical clustering has been represented using dendrograms. While effective, there certain information is lost, especially the interactions between clusters when the data are split into more than 2 parts. An alternative approach is to treat the clusters

as nodes in a graph, allowing edges to represent the strength of the connections between the clusters. This follows from the use of sparse cuts to provide an average linkage clustering shown in [94], and is extended here to formalize the HMCFP and demonstrate the utility of using graphs for the first few cuts. The differences between the HMCFP and previous work with successively cutting subgraphs are highlighted to demonstrate this utility.

Community detection has been a recent trend in graph theory, primarily arising from the field of Social Network Analysis (SNA). Maximal clique detection is one of the original NP-complete problems from Garey and Johnson, closely tied to the graph coloring problem [54]. This gave rise to the vaguely defined clustering problem of putting like nodes together while not including unlike nodes in the cluster. There has been much research into the clustering problem, with various commonly accepted heuristic algorithms such as k-means and k-medians clustering. The SNA field removed the constraint that clusters were not allowed to overlap and got away from the formalization of the clustering problem for their efforts at community detection, first popularized by Girvan and Newman [55]. Since then, there has been significant research in community detection to coarsen graphs, including a min-cut approach from Flake, et al. [47] and the Louvain method used to create cluster graphs by Blondel, et al. [21] which improved on the method from Clauset and Moore in 2004 [35].

Naturally occurring phenomena tend to follow a pattern, including the structure of graphs of connections between people, dolphins, or computer networks. They all tend to follow a power law distribution of degree [3, 14], representing a community structure of the graph. For power law graphs, the communities can be identified through the Hierarchical Maximum Concurrent Flow Problem, as a more resilient approach to missing information than many other approaches [91]. Any community seeking algorithm will tend to cut a connected community late in the identified hierarchy. The Maximum Adjacency Search is based on a preference for highly connected nodes to be visited before nodes with less connectivity. We exploit this to use it for a community seeking algorithm to reduce the size of the MCFP, while attempting to not impact the final throughput in the resulting coarsened graph. In particular, we show that the MAS approach for a graph coarsening heuristic works well in a

class of graph the exhibits high community structure, such as Random Typing Graphs.

5.2. Background

Dendrograms are often used to view hierarchical clusterings of data to see the relative closeness of each element in the data set, and the relative differences captured at each level of the cut. Dendrograms are limited in what they can display in that a single cluster is generally split into only 2 parts. While it can be split into more, the ties between the resulting clusters are lost.

Chapter 3 has a detailed discussion on the LP formulations of the MCFP and the HMCFP that we will expand on here for the purpose of graph coarsening. Both the node-edge formulation and the triples formulation imply a demand graph between all pairs i and j. Our previous work has assumed unit demand, but in this chapter we will have an explicit demand graph due to the increased demand between the (i, j) pairs as super-nodes and super-edges are created in the graph. In the general case, the demand graph can be arbitrarily populated as needed for the problem at hand. In this paper, the demand graph starts as a complete graph of unit demand that is updated during the coarsening algorithm.

5.3. Graph Coarsening Approach

We split the each set of clusters into a graph. The split into two parts is trivial. In the three part case, the split may result in a chain.

The 3-cycle is disallowed due to stability under perturbations which would lead to a chain formulation. For instance, assume the graph is separated into three parts: A,B, and C. The flow from A to B, diverted through C (and helping saturate superedges A-C and C-B) can be moved onto the superedge A-B. This reduces flow on both A-C and C-B by the same amount. The flow from B to C diverted through A can then be sent directly along B-C, reducing the flow on A-B below its capacity. At this point, any remaining flow along A-C can be diverted to the A-B-C path and therefore eliminate the A-C superedge. Otherwise there would be additional throughput. In the event that this cannot happen, the next cut

should be a tie for the same throughput, resulting in a 4-(or more)-cycle.

In the 4 part case, the options are a chain, a cycle, or a star. The "triangle with a tail" case is ruled out for the same reason that the 3-cycle is disallowed. In fact, any result must be triangle-free, for the same rationale given above.

Triangles may exist when there are ties for the cut. However in a case where the graph is robustly split into multiple parts, the cannot exist, as shown in the above proofs. The 5 part case is where the results get interesting. It was shown in [98] that finding the sparsest cut is not guaranteed to be found by a polynomially bounded formulation. For the cycle, star, and chain cases, a polynomially bounded formulation works, but this is not known before running the algorithm. The 5 part case may also result in a $K_{3,2}$, which represents the smallest case of the polynomially bounded formulation failing to find the sparsest cut.

In the cases of 5 or more partitions at a given cut without ties, it may be argued that this represents sufficient homogeneity in the data to stop attempting to split the nodes from each other at that level. A 5 (or more) part case with ties or for cases with less than 5 partitions, the data still represent different clusters and the hierarchical partitioning should continue.

The Maximum Adjacency Search (MAS) was introduced in 1993 as a heuristic for the edge connectivity problem [95] and used to partition graphs later that year [28]. It was also modified to find the minimum cut in graphs by Stoer and Wagner in 1997 [117]. This min-cut approach was added to the Boost C++ library in version 1.45.0 and later modified by the authors of this paper to expose the underlying maximum adjacency search algorithm and to use a visitor pattern in version 1.53.0 The nature of the search algorithm tends to visit nodes within a given community before expanding to a new one. This coincides with the tendency of sparse cuts to not split communities in a graph, making it a reasonable choice to explore for a heuristic.

Since the MAS is not as widely known as depth-first search or breadth-first search, we provide a refresher for the unweighted graph case here. The MAS initializes each node to 0 "reach count", which will be tracked as a level function when each node is actually visited and marked as such. Beginning with a node in the graph, increment the reach count of

all of its neighbors. Then select the next unvisited node with the highest reach count, ties broken arbitrarily. Again, increase the reach count on each unvisited neighbor. Then repeat the process until every node in the graph has been visited. Previous work has labeled the edges with the final reach count that they provide between the two nodes at each end of the edge. The labels can be used to create n-connected edge disjoint paths between nodes. In [28], they track the MAS level at which each node is visited and then use the resulting level function to identify communities during the forward pass. The authors of [28] also use a backward pass to identify graph partitions, which we do not use in this paper.

We created an MAS visitor for community detection to coarsen the graph before running the LP to speed the processing of the graph. We use the visitor pattern exposed by the Boost Graph Library to monitor the cut weight when visiting each node. That is, we track the total edge weight to separate all visited nodes (including the current one) from all unvisited nodes. Since this approach is being used on capacitated graphs, we use the edge capacity for the edge weight in this context. The cut weight tends to fluctuate as the graph search continues, beginning low, increasing or remaining constant within a highly connected group of nodes, then decreasing or remaining constant as some additional group members get added, finally increasing again when leaving that group to start a new one. Following this trend, we start tracking a community when the cut weight first decreases, then declare a community found if the cut weight increases again.

To minimize the impact of the choice of starting node, we repeat the MAS after reinitializing the reach counts starting from each node. We exclude a node of maximum degree and its immediate neighbors as starting nodes, to attempt to steer the algorithm to find communities other than the obvious one. After an iteration of the MAS starting from all allowable nodes, the "best" cut is the one closest to combining half of the nodes in the graph, in order to reward finding a larger community. Once again, there may be ties where multiple communities are found of the same size, where we break these ties abritrarily.

The resulting community is contracted into a super-node by combining the nodes within the community into a single node. Any cut edges to the same node outside of the community

are combined into a super-edge with the resulting capacity being the sum of the capacities of the combined edges. As mentioned in Section 5.2, the demand graph begins with unit demand. When contracting a community into a super-node, the demands are updated similarly to the capacities, so that the demand crossing the cut to separate the community is updated using the weight of the super-node to ensure that the resulting maximum concurrent flow calculation is unaffected by the coarsening. Any community detection algorithm will have a low likelihood of detecting a community that would be separated by a sparse cut. Therefore, the coarsening approach used here should not often impact the resulting maximum concurrent flow. Our approach is summarized below.

Graph Coarsening Using MAS

```
Procedure Find_Community_With_MAS:
  Initialize MAS reach count at each node to 0
  Initialize the set of unvisited nodes to the set of all nodes
  Set the reach count of the start node to 1
  While the set of unvisited nodes is not empty:
    Let i = the node in the set with the largest reach count
    Increase the reach count of all neighbors
    Mark i as visited and remove it from the set of unvisited nodes
    If the cut weight has decreased this iteration after decreasing earlier:
      Return the set of visited nodes (including i) as a found community
  If all nodes have been visited without finding a community:
    Return an empty community

Coarsen_Graph:
  Repeat
    For each node v in the graph:
      c = Find_Community_With_MAS with v as a start node
      Let c_max be the largest c found this pass through the graph

    If c_max was found this pass:
      Contract all nodes in c_max to a super-node

  Until c_max is not found
```

5.4. Experimentation Approach

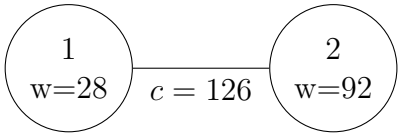
As discussed in Chapter 2, we use over 6000 random bipartite graphs with up to 70 nodes in each partition and variation in the probability of an edge, 500 random geometric graphs on the unit square, 400 random geometric graphs on the unit sphere, and 100 random typing graphs as synthetic social networks with an average of 140 nodes and 346 edges to compare the LP reduced triples formulation of the MCFP before and after coarsening the graph. We compare the throughput values calculated as well as the time and memory utilization for the calculations. We also use some selected graphs from previous works in the MCFP area [19, 91] for specific illustrations.

With the exception of the NCAA 2004-2006 graph from Mann [91], all graphs used in the analysis were created with unit demand and unit capacity. This graph used the number of times that teams played each other as the capacity, and had unit demand regardless of the number of games played.

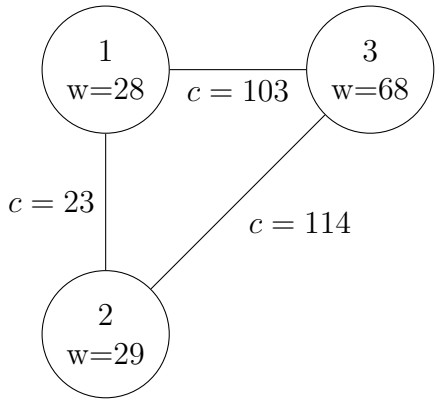
5.5. Discussion

As an example, we show the first few clusters of the NCAA data from Mann, et. al. [90]. For comparison, we also show the results from that study. Both sets of results are shown until the graphs are separated into five supernodes.

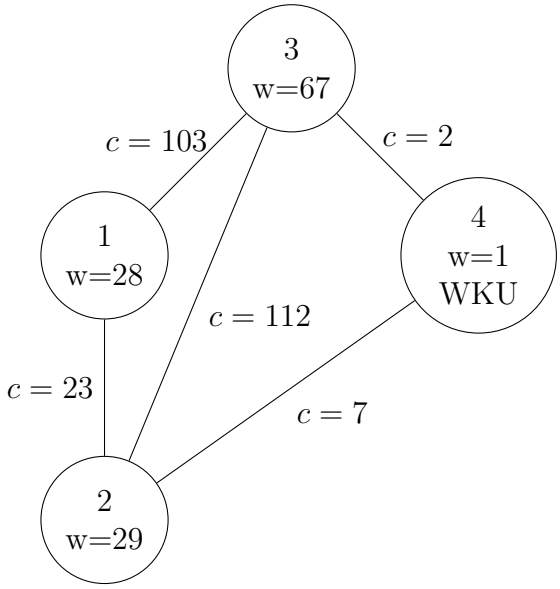
We see some key differences in the results based on the hierarchical (Figure 5.1) nature as opposed to the divisive (Figure 5.2) nature of the algorithm used. First, the divisive formulation has a tie in the third cut, finding five supernodes earlier than in the hierarchical formulation. In particular, Western Kentucky is separated from the rest of the nodes in the third cut in the Hierarchical formulation. This is contrast to the Divisive approach where Western Kentucky is not separated until the ninth cut. In this case, Western Kentucky remains in a supernode consisting of it, the SEC, Sunbelt, and Conference USA.



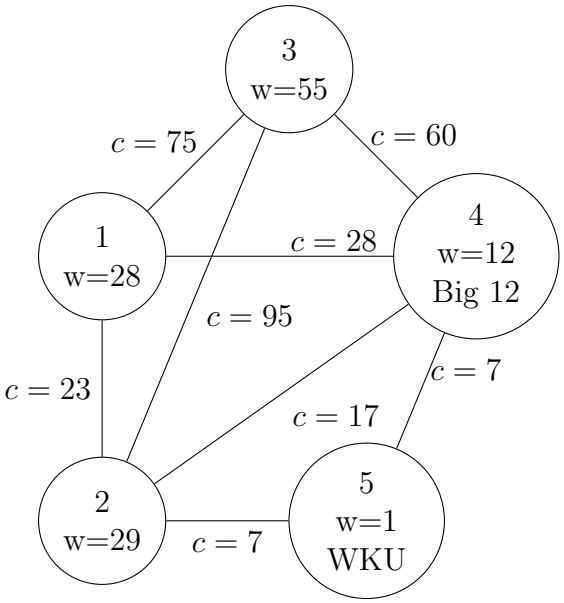
(a) Cut 1: z=0.048913



(b) Cut 2: z=0.0839461



(c) Cut 3: z=0.134328



(d) Cut 4: z=0.159091

Figure 5.1. HMCFP Results on 2004-2006 NCAA data

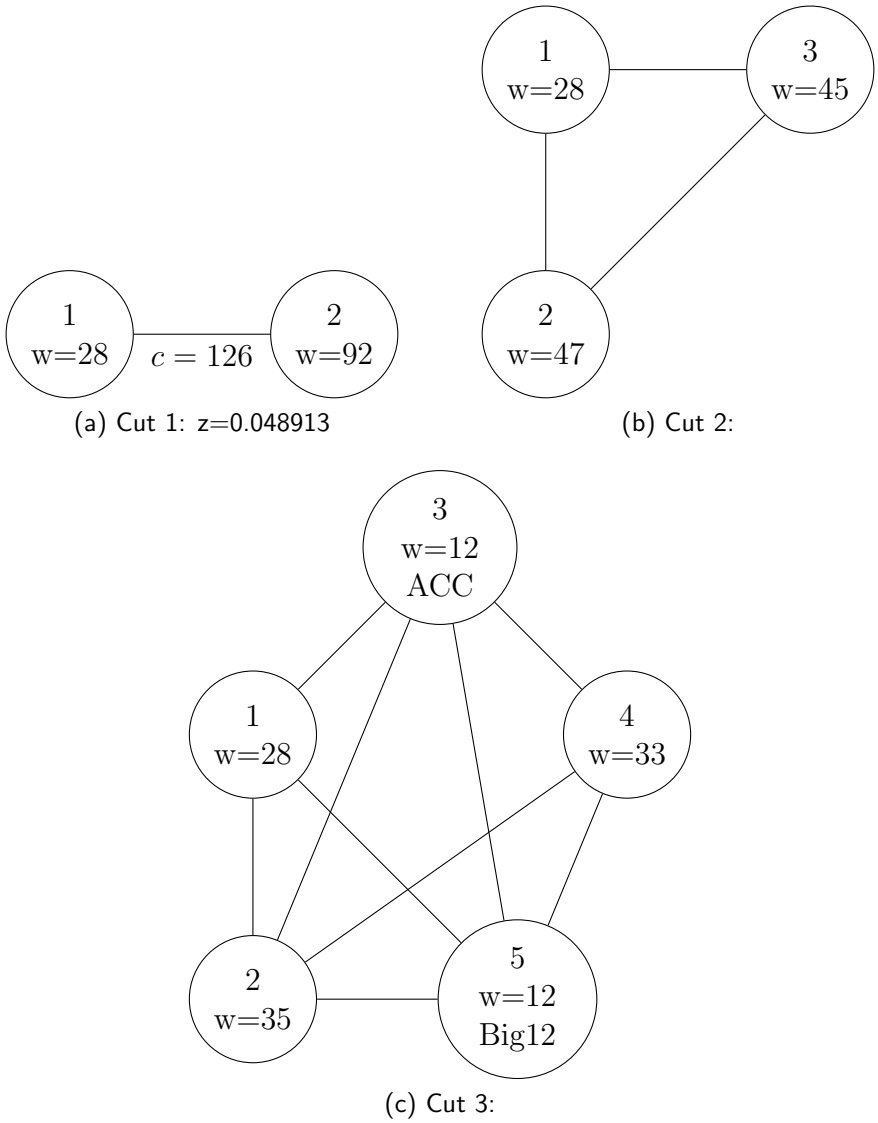


Figure 5.2. Divisive MCFP Results on 2004-2006 NCAA data

This difference is due to the hierarchical form better modeling the interactions from out of conference games. At the time these data reflect, Western Kentucky had just moved into Division 1-A and was independent. It was also attempting to join the Sunbelt Conference, yet had not done so. In the divisive form, some interactions are not captured due to being previously separated from that portion of the graph. In this case, the divisive case ignores the interactions between WKU and 2 MAC teams: Ball St. and Bowling Green. The hierarchical form allows WKU to be separated early when these 2 edges "pull" it from the supernode

containing the SEC (3 edges) and Sunbelt (4 edges) conferences. A supernode containing those 2 conferences exists in both formulations of the problem. We explore the performance of the graph coarsening approach on this graph in comparison to the HMCFP in the next section, after the discussion on the Biswas graphs.

5.5.1. Selected Example Graphs

The 20 node graph from Biswas [19] is shown in Figure 5.4, before and after coarsening. We show only the first round of coarsening to illustrate the results of the process. In this case, the communities are obvious to the eye and the graph is small enough to check the results by hand. The remaining iterations of coarsening are not shown, but function as expected to create the 5 node trestle graph shown in Figure 5.3, while maintaining the same MCFP throughput of 0.03125. Since the trestle graph has multiple optimal solutions for the MCFP (and for the SCP) but does not result in a gridlock style cut, the community detection algorithm should, and does, fail to find a community after the 5 super-nodes have been created. After coarsening from 20 nodes to 5 super-nodes, the resulting MCF finds the same solution as the uncoarsend version, with the same throughput and cut edges. The fact that the algorithm detects all 5 communities and does not detect non-existent communities is evidence that it is an efficient method of coarsening graphs with high modularity.

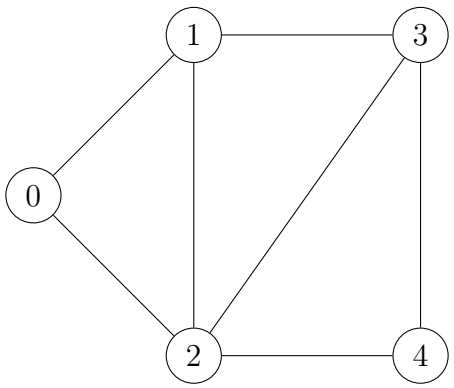
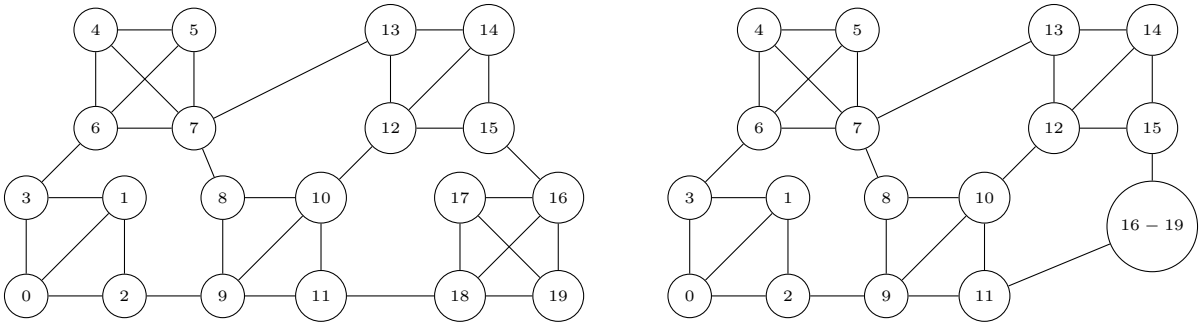


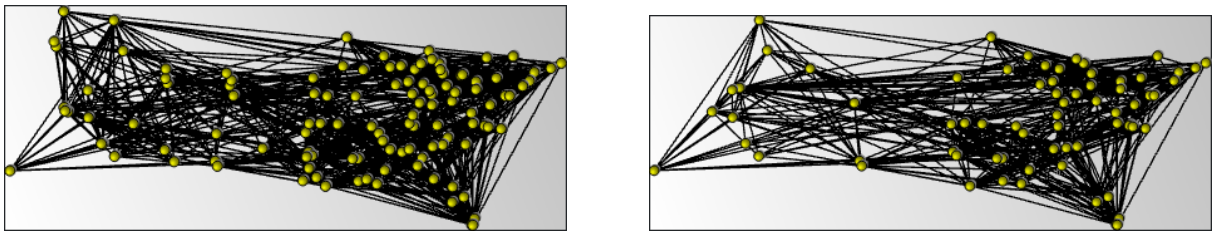
Figure 5.3. 5 node trestle graph



(a) Original Graph (b) After 1 round of coarsening

Figure 5.4. 20 node graph from Biswas

We see similar impacts from coarsening the NCAA graph from Mann [91] shown in Figure 5.5. This graph was too large to calculate by hand, but the throughput value is maintained. We can see in the figure that teams near each other in the same conference (and division) are combined, while nearby teams in different conferences (e.g. MWC, PAC-10, C-USA) are not contracted into a super-node. This is especially obvious in the western part of the country, where the sparsity of nodes is easier to see. In general, the community structure is found, though the full 6-clique within each division may not have been found. For instance, Washington and Washington State were combined, along with Oregon and Oregon State. However, the 4 teams were not contracted into a single super-node. This normally occurs when a node is "pulled" out of the community by a set of strong connections to outside communities due to bowls or independent teams. In particular, Notre Dame as an independent team has been shown to have a profound effect on the community structure detected by various algorithms on this graph. During the time of the data captured for this graph, Western Kentucky (WKU) had just joined Division 1-A from the 1-AA OVC, and was playing as an independent team before joining the Sunbelt Conference. This caused portions of the Sunbelt Conference to be pulled out early, with WKU as a central node connecting those teams to the rest of the graph. The graph was reduced from 120 nodes to 57 nodes, giving a runtime that was 10% of the original graph taking 20 seconds rather than 200 seconds to solve.

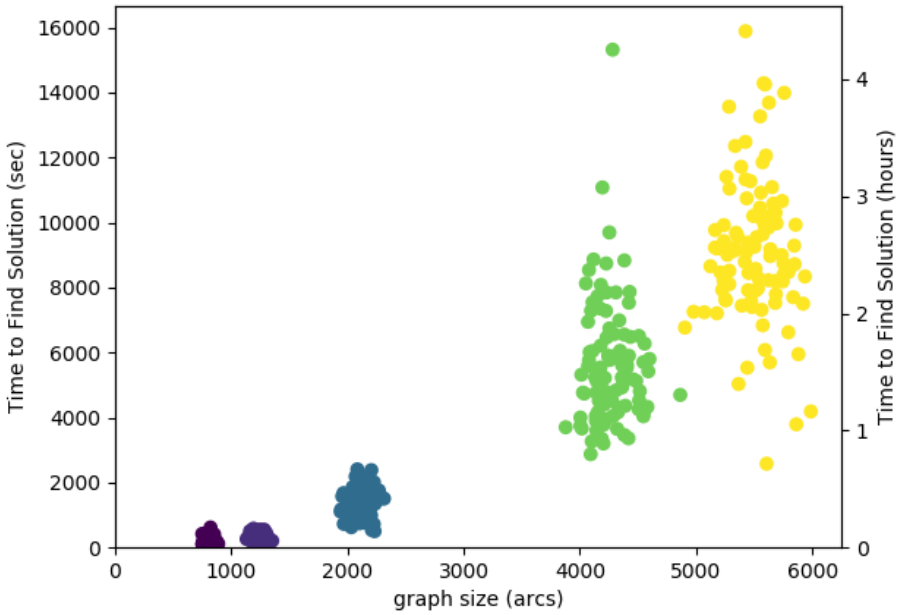


(a) Original Graph (b) After Several Rounds of Coarsening

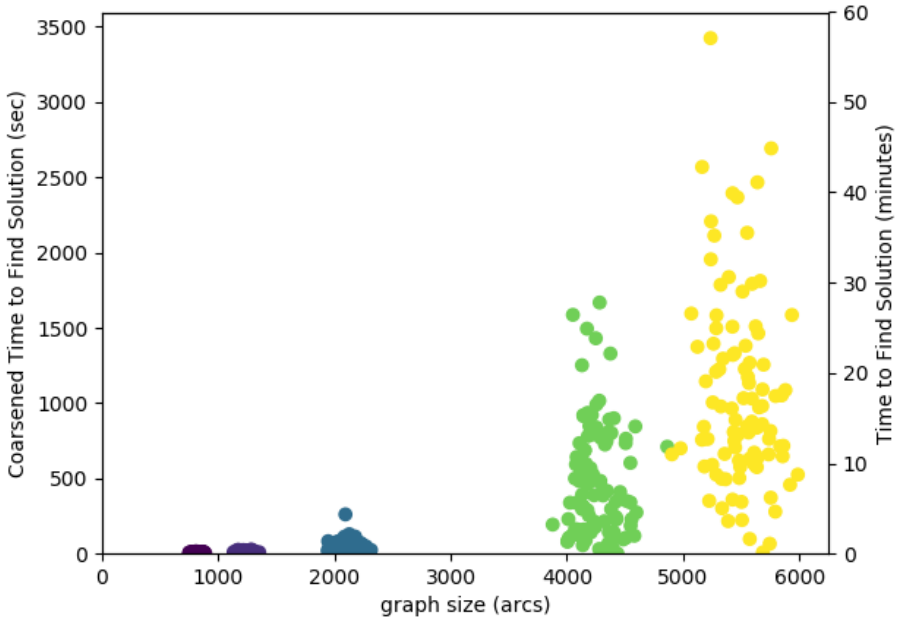
Figure 5.5. NCAA 2004-2006 data

5.5.2. Results on Random Geometric Graphs

On the set of random geometric graphs on the unit square, the results of coarsening with the MAS heuristic show a similar 90%+ reduction in runtime to calculate in Figure 5.6, and roughly 50% reduction in memory utilization in Figure 5.7. However, there is a penalty in the accuracy ($error = Z_{mas} - Z_{MCFP}$) of the results, shown in Figure 5.8, either due to communities spanning the sparsest cut, or due to the inclusion of communities within other larger communities. The likelihood of finding the correct answer decreases as the graph gets denser. Since the number of nodes is constant for all RGGs on the unit sphere, the density increase is completely explained by the larger threshold radius resulting in more edges in the graph. Not only does the likelihood of error in denser graphs increase, but also the magnitude when there is an error. The average error grows super-linearly, even as a percentage of the correct value. This growth indicates that this coarsening approach may not be ideal for this particular class of graph, due to the low modularity of most of the results, though higher than in random graphs.

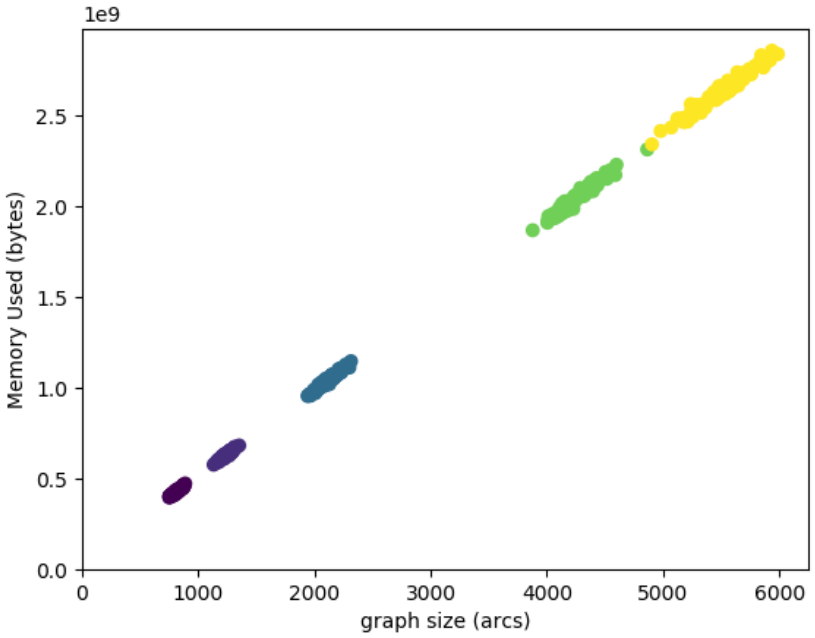


(a) LP time

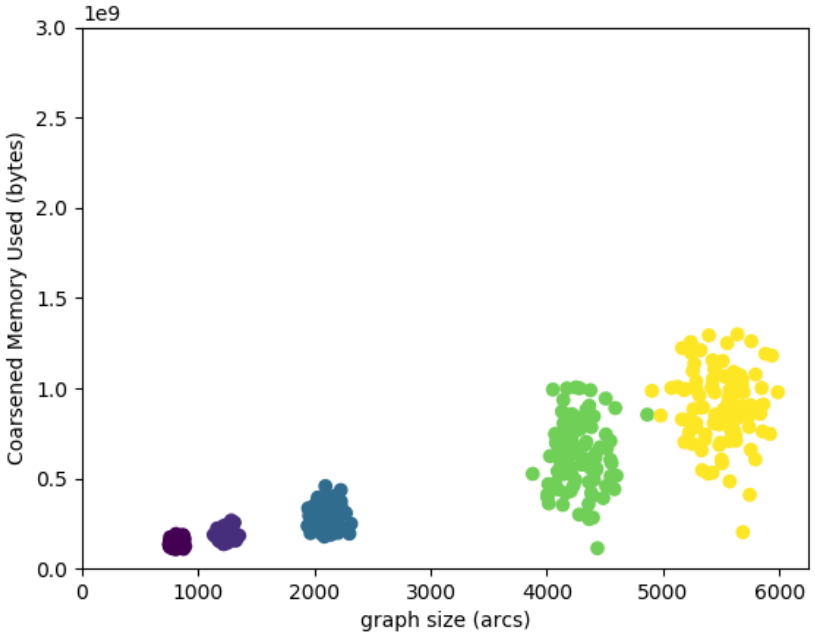


(b) time after coarsening

Figure 5.6. Comparison of runtime for the RGG on the unit square

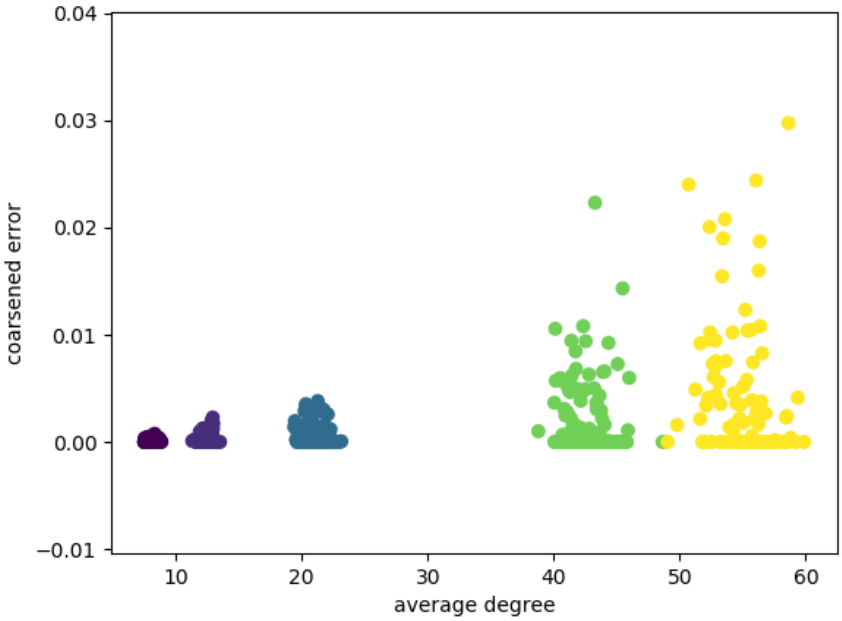


(a) LP mem

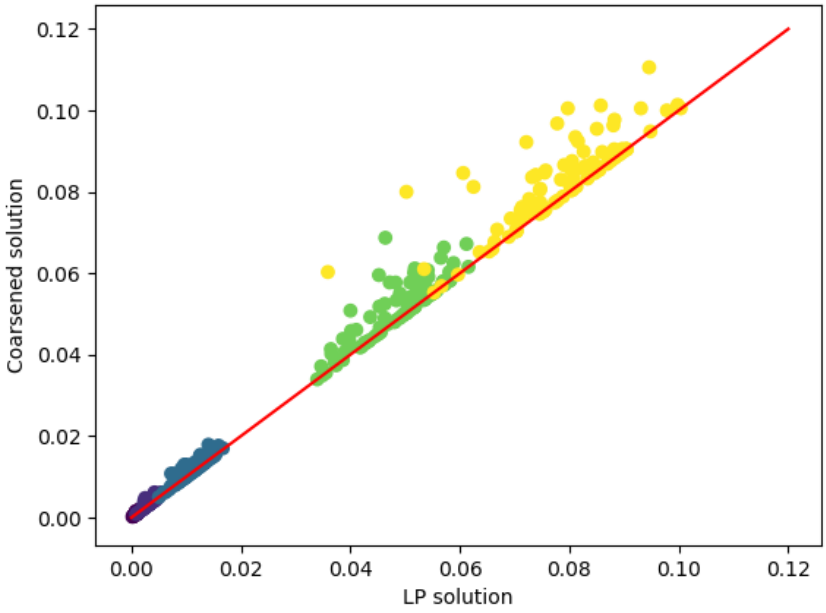


(b) mem after coarsening

Figure 5.7. Comparison of memory usage for the RGG on the unit square



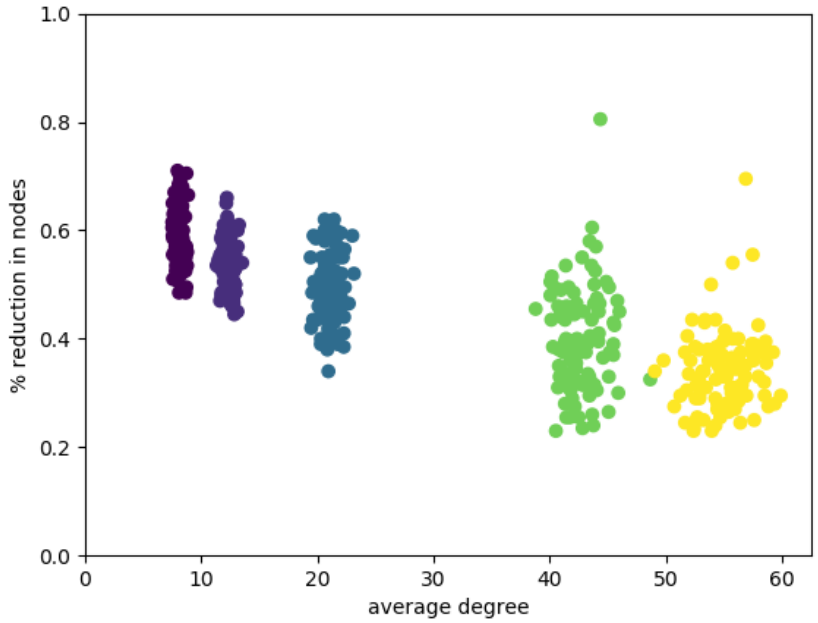
(a) Error due to coarsening



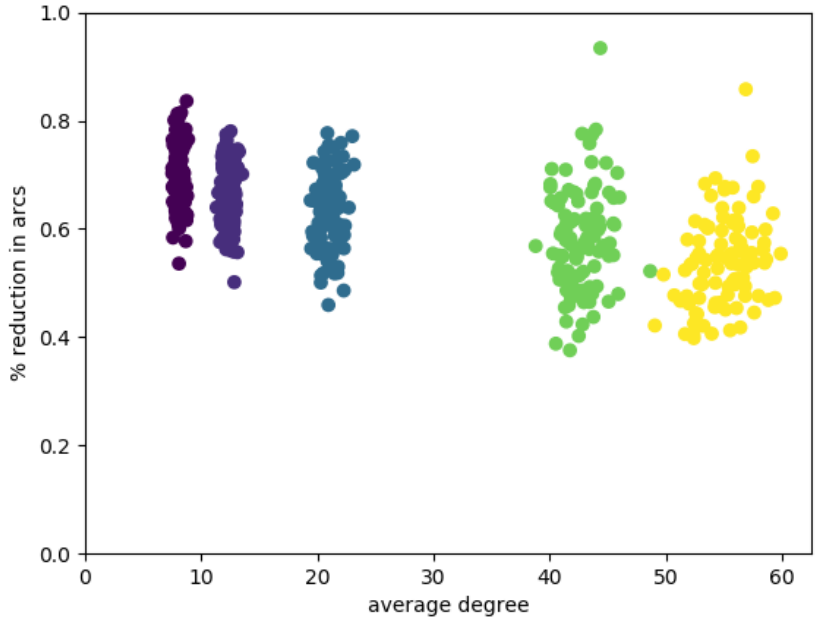
(b) solution comparison

Figure 5.8. Comparison of accuracy for the RGG on the unit square

We looked into some reasons for the reduced memory and time utilization in Figure 5.9. The number of nodes and edges are both reduced by about half by contracting communities into super-nodes, in both cases, diminishing as the graph density increased. This seems to be related to the reduced modularity of denser graphs, where in sparser graphs any nearby group of nodes would be an obvious community with limited connectivity to outside groups.



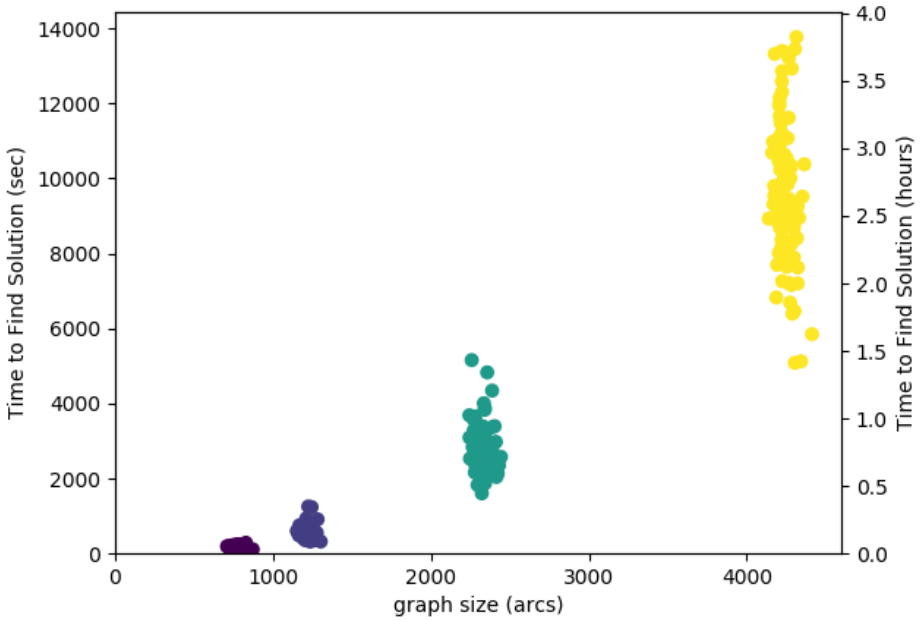
(a) pct node reduction



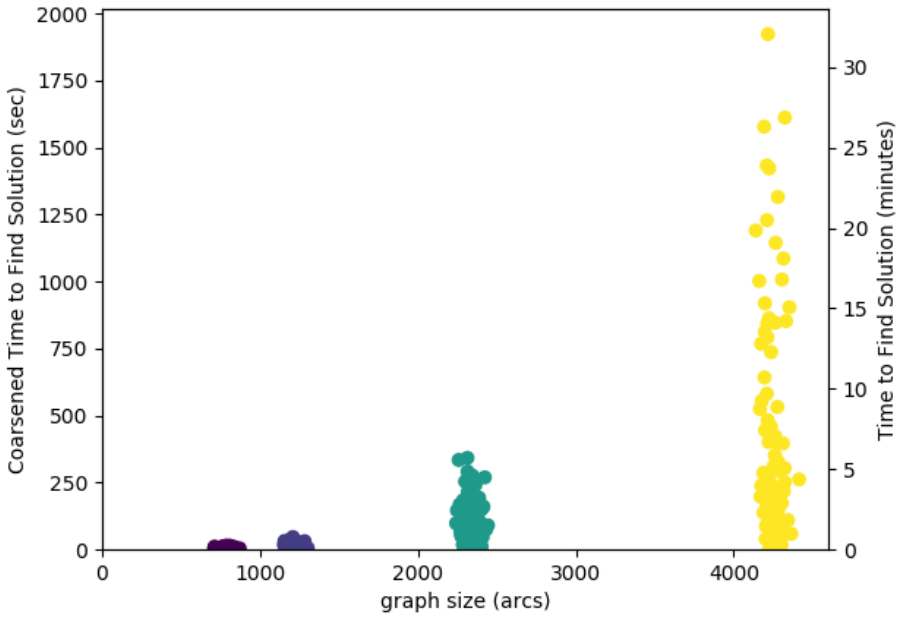
(b) pct edge reduction

Figure 5.9. Reduction in nodes and edges for RGGs on the unit square

Similar to the RGGs on the unit square, those on the unit sphere exhibit a reduction in processing time of 90%+, shown in Figure 5.10 and a reduction in memory utilization of roughly half, shown in Figure 5.11. We also see a similar error after coarsening to the cases on the unit square, where the solution is more likely to be correct in the sparser cases but growing as the density increases and the modularity decreases, shown in Figure 5.12.

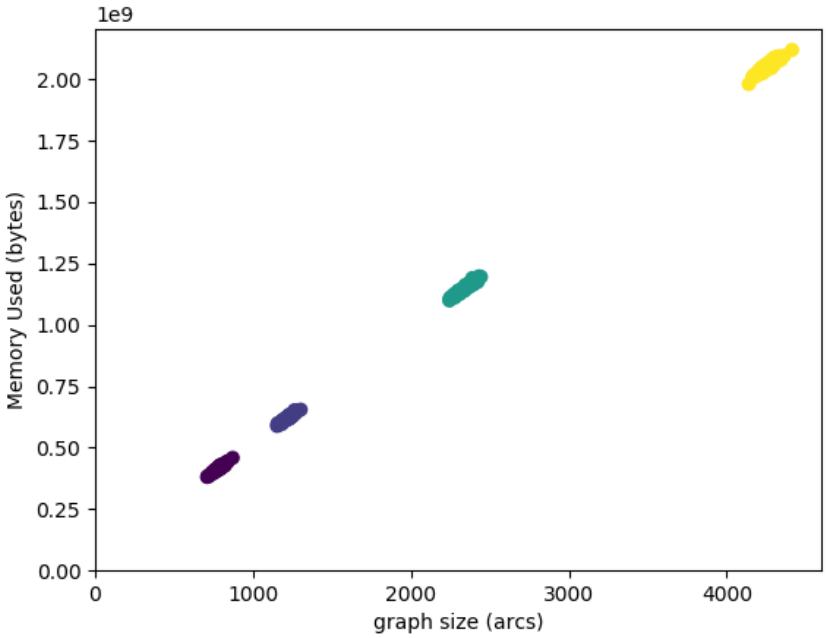


(a) LP time

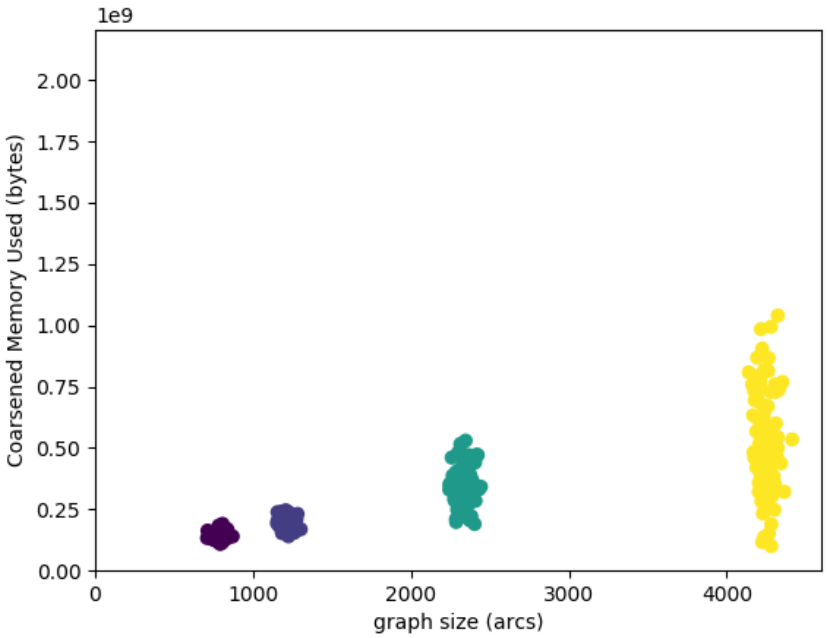


(b) time after coarsening

Figure 5.10. Comparison of accuracy for the RGG on the unit sphere

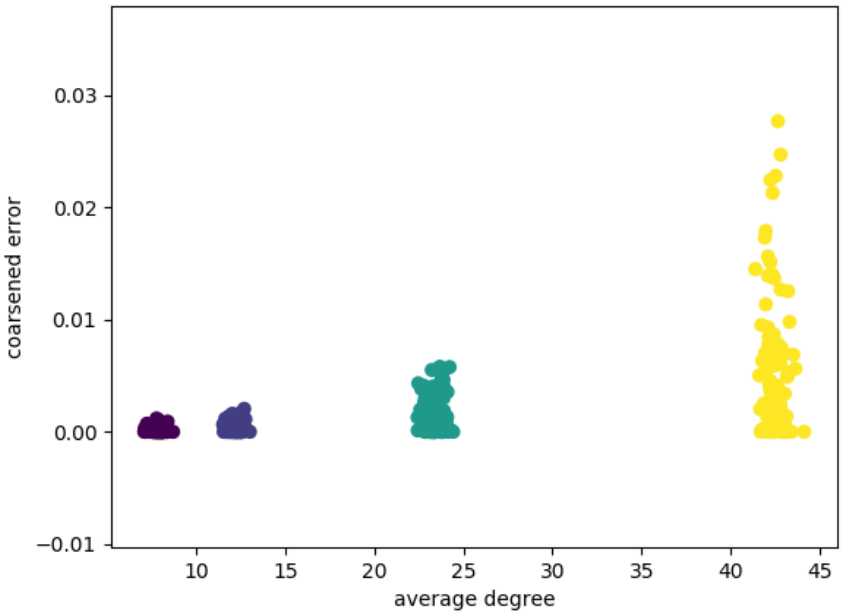


(a) LP mem

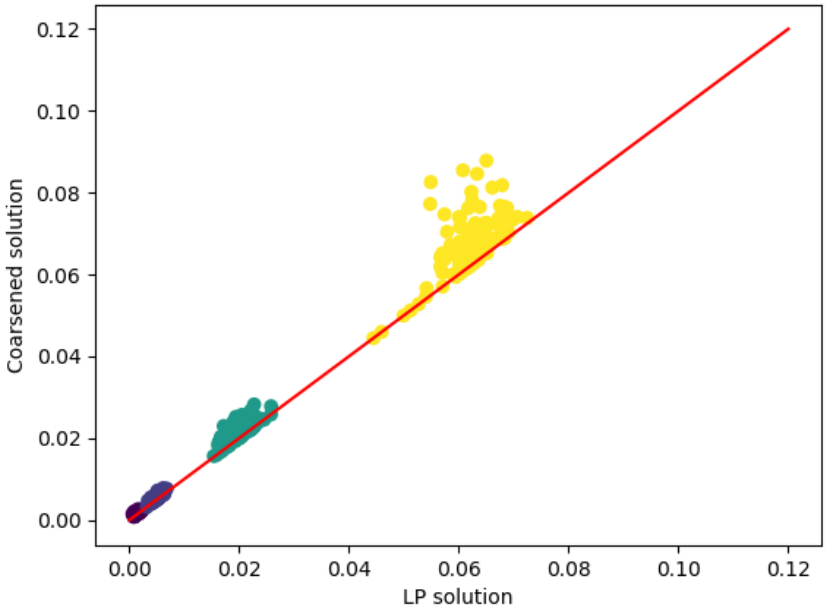


(b) mem after coarsening

Figure 5.11. Comparison of memory usage for the RGG on the unit sphere



(a) Error due to coarsening

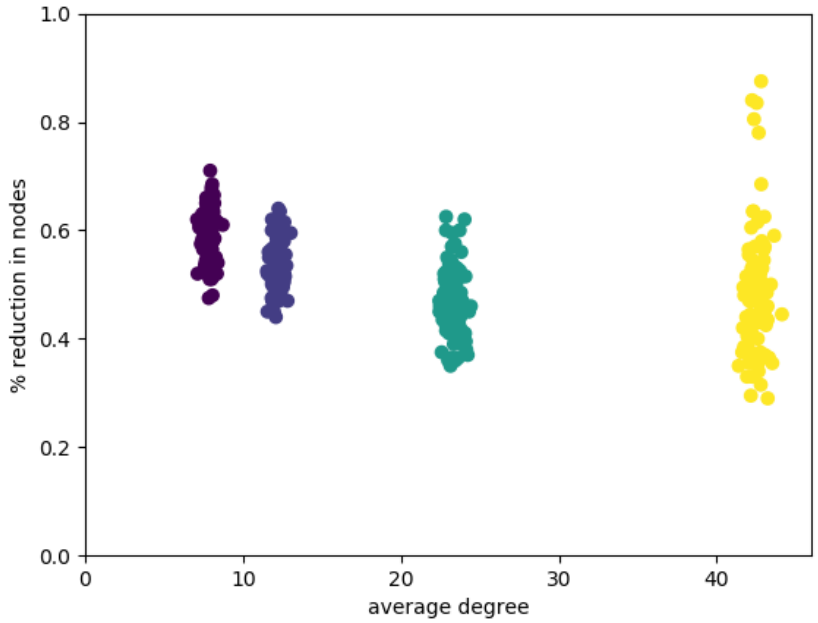


(b) solution comparison

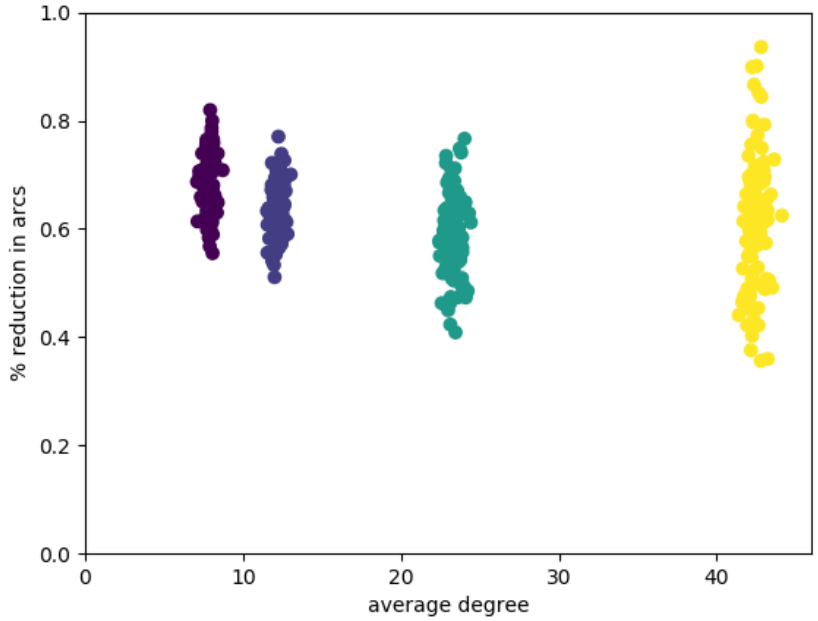
Figure 5.12. Comparison of accuracy for the RGG on the unit sphere

On the unit sphere, we see in Figure 5.13 that the final number of nodes and edges is reduced by about half in both cases. The denser cases also have more variability than on the unit square. The average degree also exhibits reduced variability compared to graphs on the unit square, due to the homogenizing effect of removing the boundary conditions on the topology.

We see that the reduced graph size in both nodes and edges on the unit sphere does not drop as low as in the case of the unit square and exhibits more variability as a function of average degree. The boundaries of the unit square tend to bias results by reducing the number of possible neighbors for connection as a node approaches a boundary, or worse a corner. The unit sphere avoids this case, and when generated uniformly over the sphere also avoids any polar bias that might arise. The flatter results for percent reduction were expected due to the removal of boundary conditions, but the increase in variability was not.



(a) pct node reduction

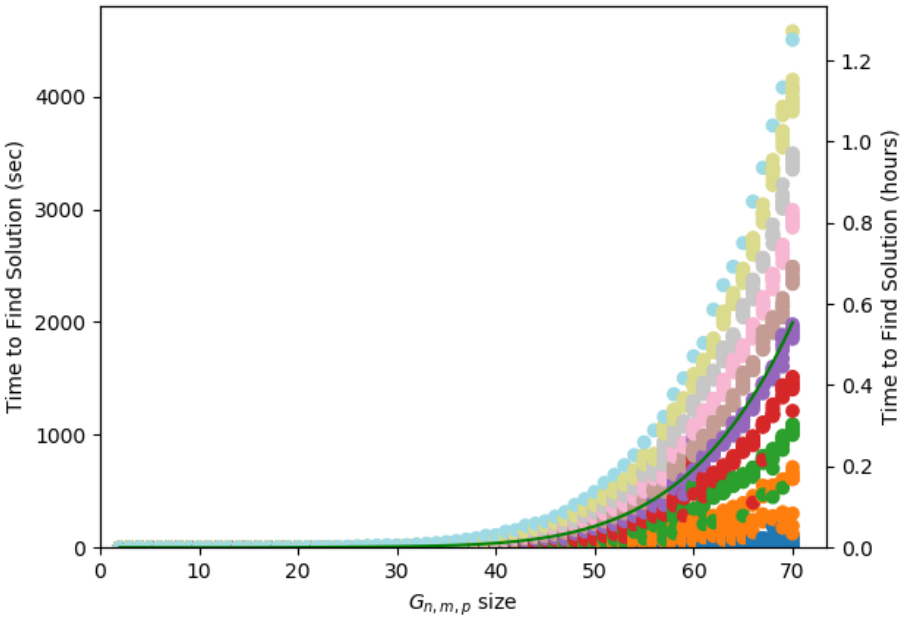


(b) pct edge reduction

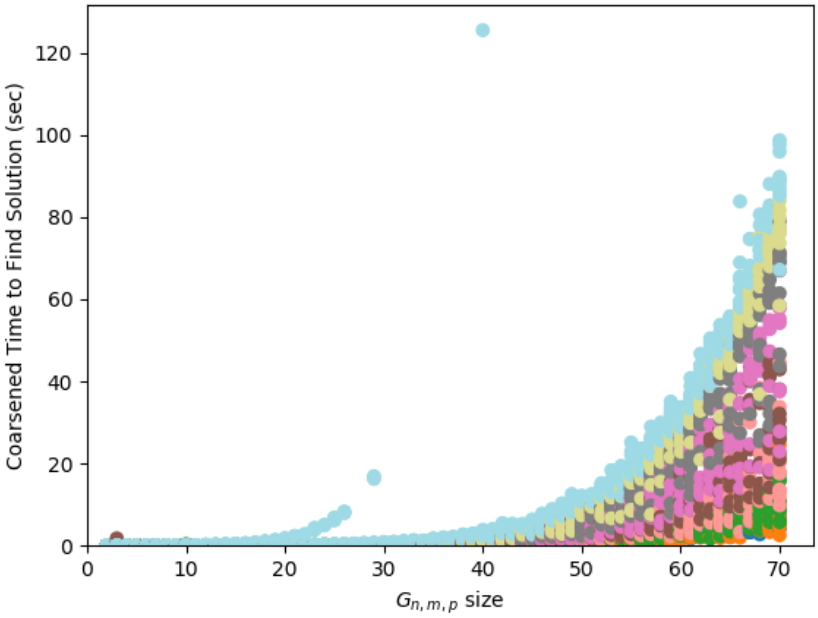
Figure 5.13. Reduction in nodes and edges for RGGs on the unit sphere

5.5.3. Results on Random Bipartite Graphs

The random bipartite graphs show an even greater improvement in processing time than the RGGs, with a reduction of over 95% in all but a few cases, shown in Figure 5.14. We also point out the order of magnitude decrease in scale from $1e9$ to $1e8$ after contraction. The cases that did not show benefit were in a few of the cases with the probability of an edge being 0.9 and having limited, if any, community structure. In particular a case with 40 nodes on either side resulted in no communities detected, therefore no coarsening. This case can be seen at the top of the post-coarsening graphs in Figure 5.14 and Figure 5.15. We also see in Figure 5.16 that coarsening is prone to error, especially in the cases of weakly connected graphs of small minimum degree. This is not the case in disconnected graphs, since the MAS approach will drop the cut weight to 0 before jumping across the gap to another component. The error introduced asymptotically approaches a limit from above that matches bounds presented in previous work [119].

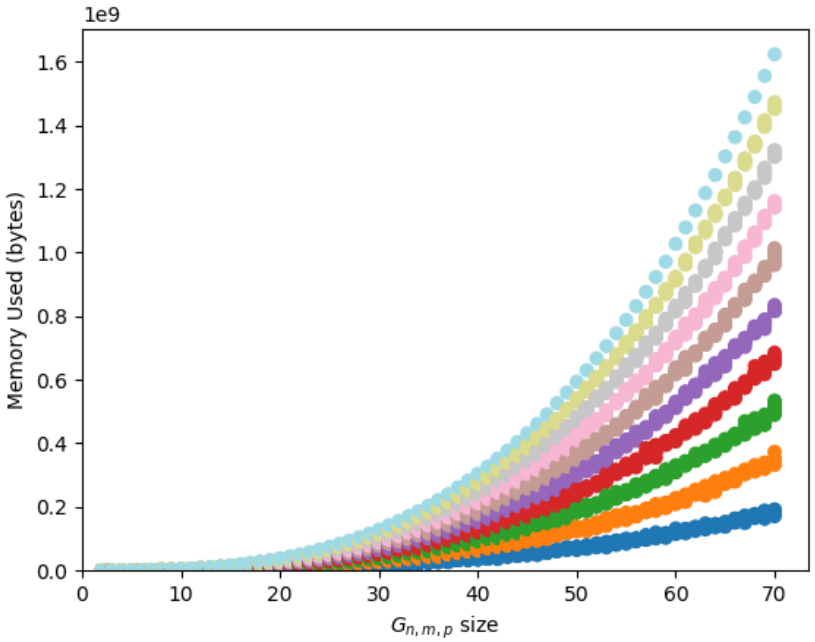


(a) LP time

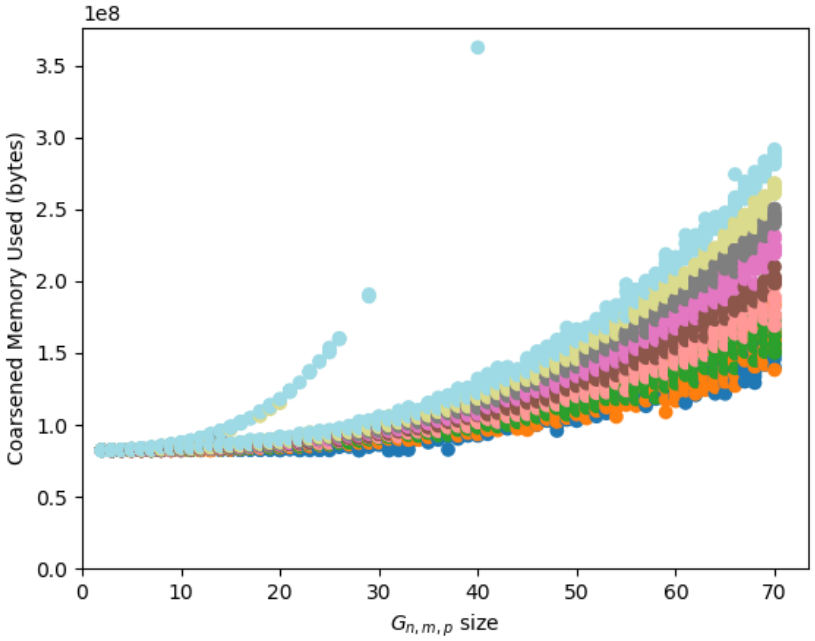


(b) time after coarsening

Figure 5.14. Comparison of runtime for random bipartite graphs

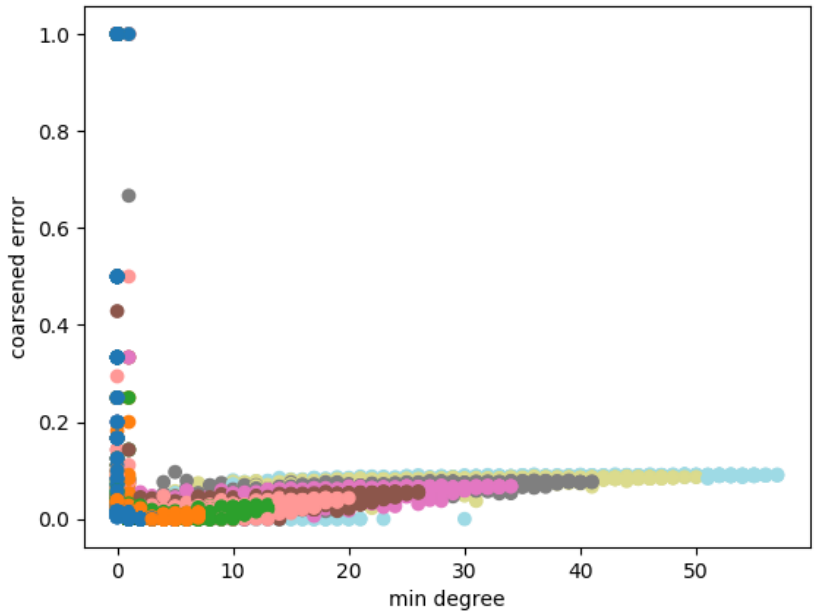


(a) LP mem

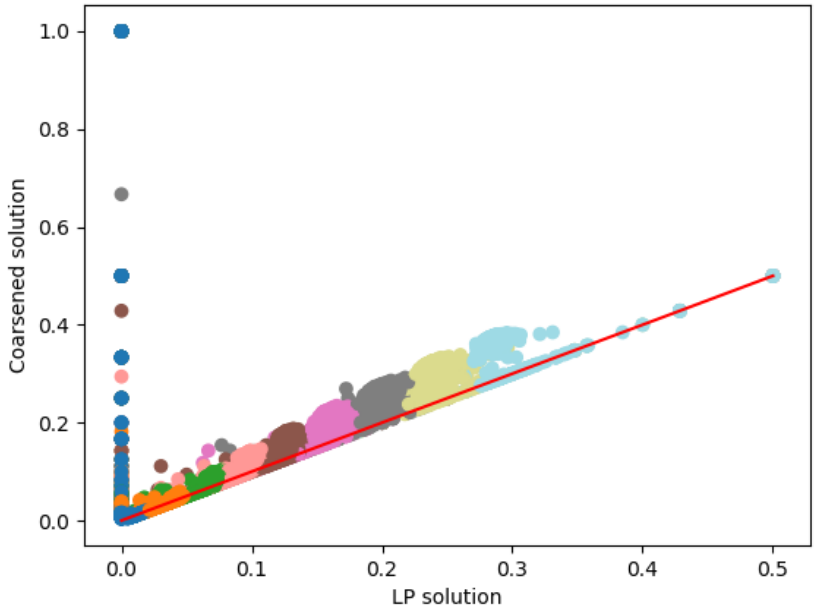


(b) mem after coarsening

Figure 5.15. Comparison of memory usage for random bipartite graphs



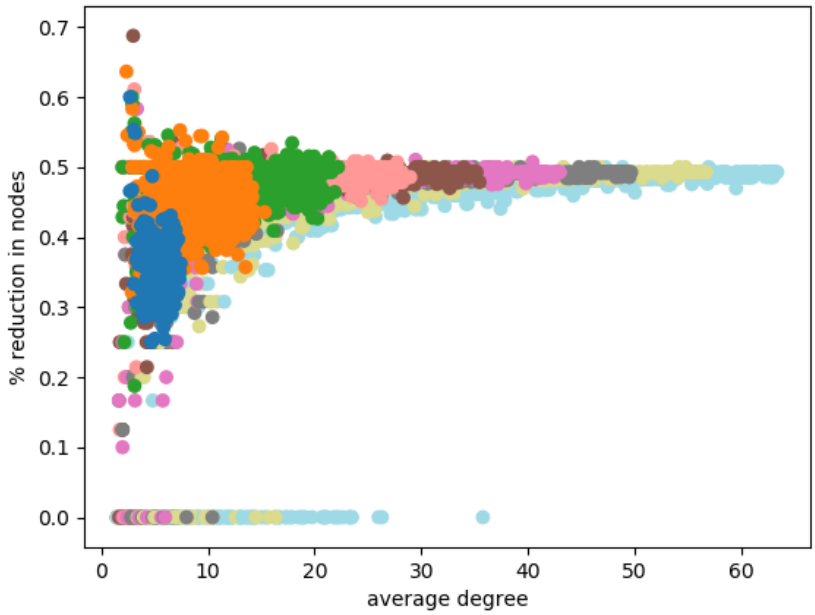
(a) Error due to coarsening



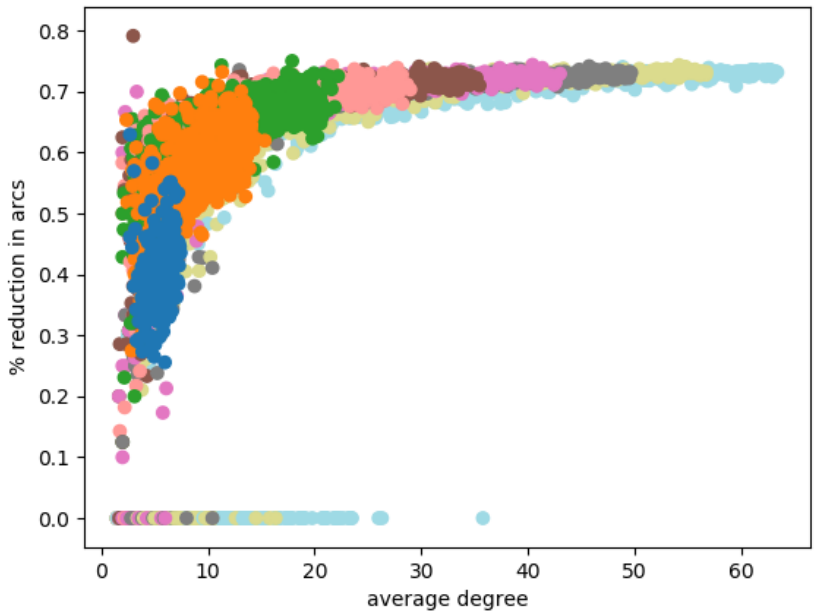
(b) solution comparison

Figure 5.16. Comparison of accuracy for random bipartite graphs

Other than the few cases where no coarsening was obtained, we can see in Figure 5.17 that the reduction asymptotically approaches 50% from both sides as the average degree increases. The bipartite graphs can be divided into 2 groups: a and b. Several nodes from the same group, say b, are contracted into super-nodes. As the graph becomes denser, the algorithm from a start node in a will add most of the nodes from b. In the limit, it will capture all nodes from b and one node from a, creating a graph half the size of the original.



(a) pct node reduction

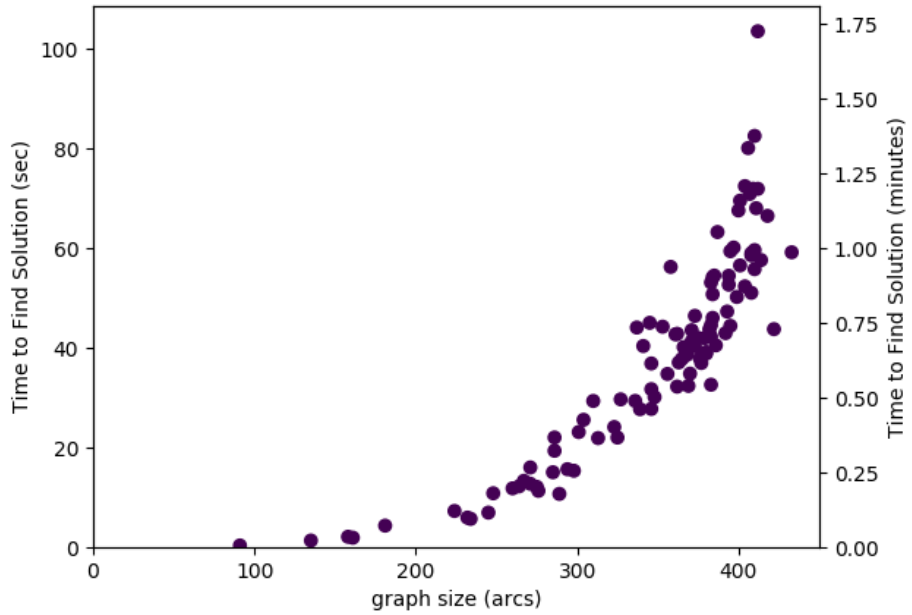


(b) pct edge reduction

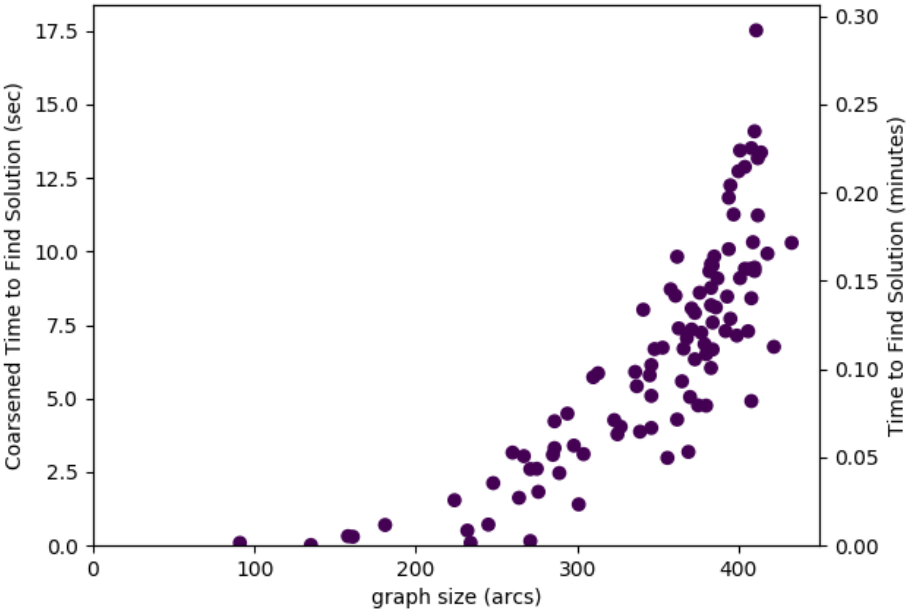
Figure 5.17. Reduction in nodes and edges for random bipartite graphs

5.5.4. Results on Random Typing Graphs

On RTG, with a higher modularity than the other classes of graph in this paper, still exhibit the a 90%+ reduction in time (Figure 5.18) and 50% reduction in memory (Figure 5.19) similar to the other sections. While there are still errors, they are fewer and lower magnitude than other classes of graphs in this paper, shown in Figure 5.20. This class of graph almost always has a min degree of 1, due to the nature of the generation algorithm. However, the reduction in memory and runtime come from the contration of communities into super-nodes. Graphs, real or synthetic, with high modularity seem to be good candidates for this heuristic approach, based on the results shown here.

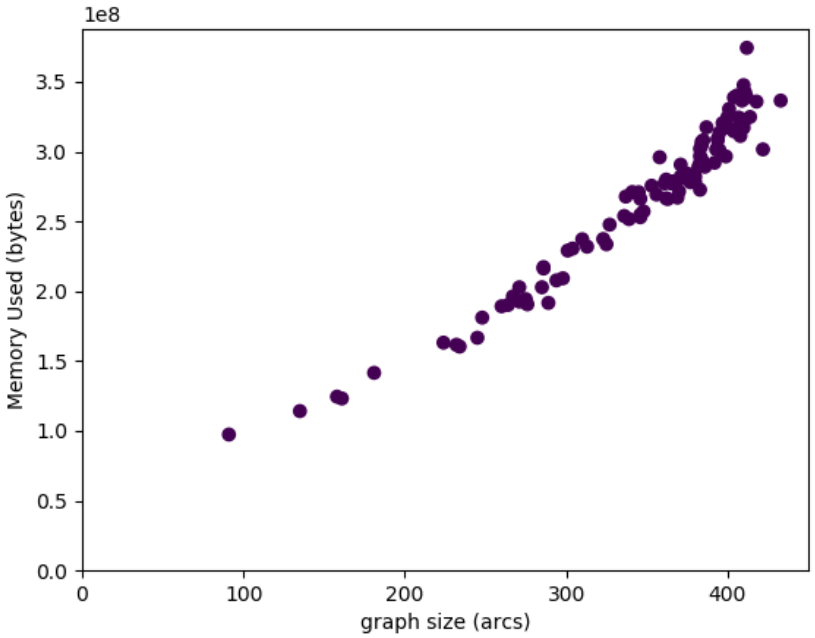


(a) LP time

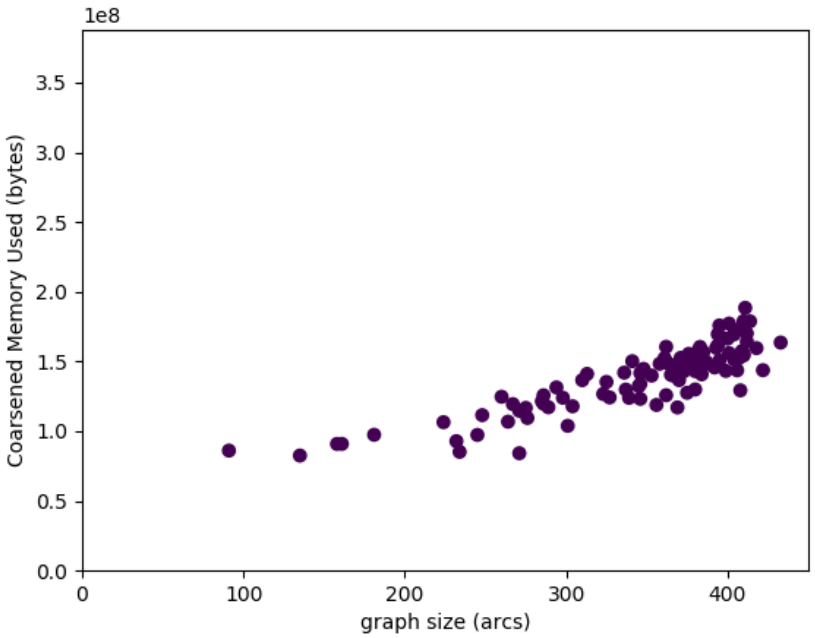


(b) time after coarsening

Figure 5.18. Comparison of runtime for random typing graphs

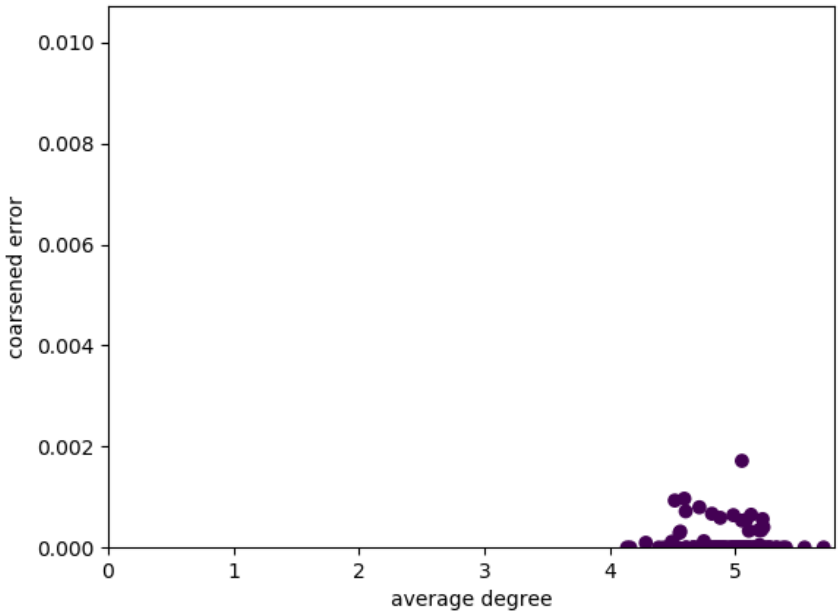


(a) LP mem

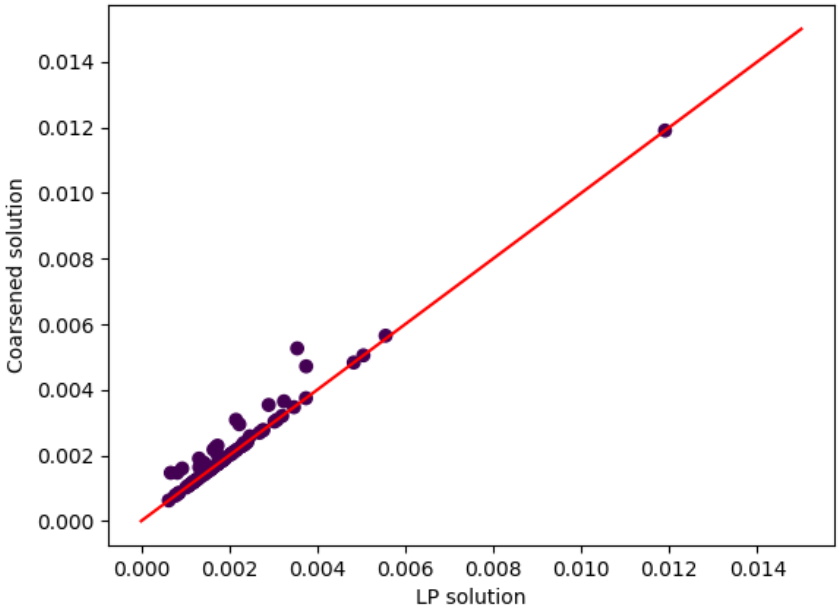


(b) mem after coarsening

Figure 5.19. Comparison of memory usage for random typing graphs



(a) Error due to coarsening

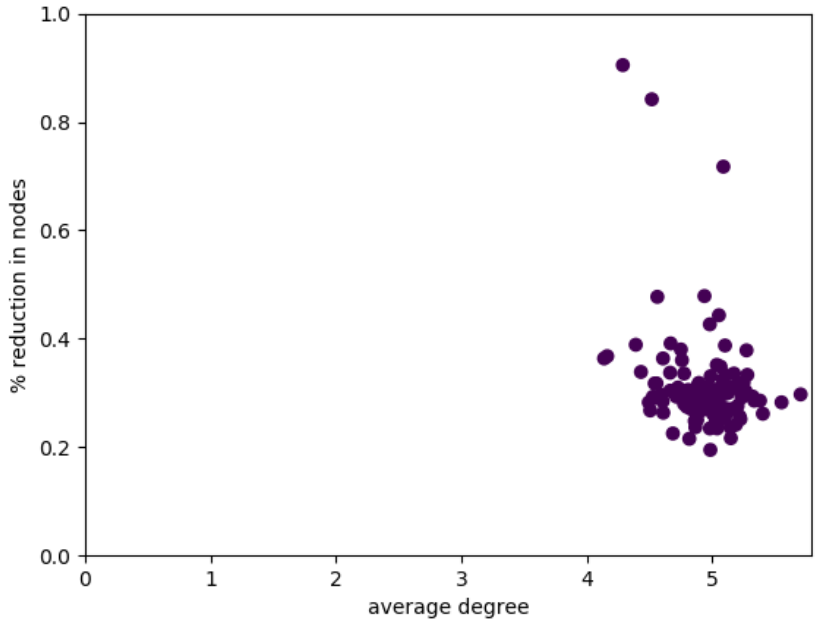


(b) solution comparison

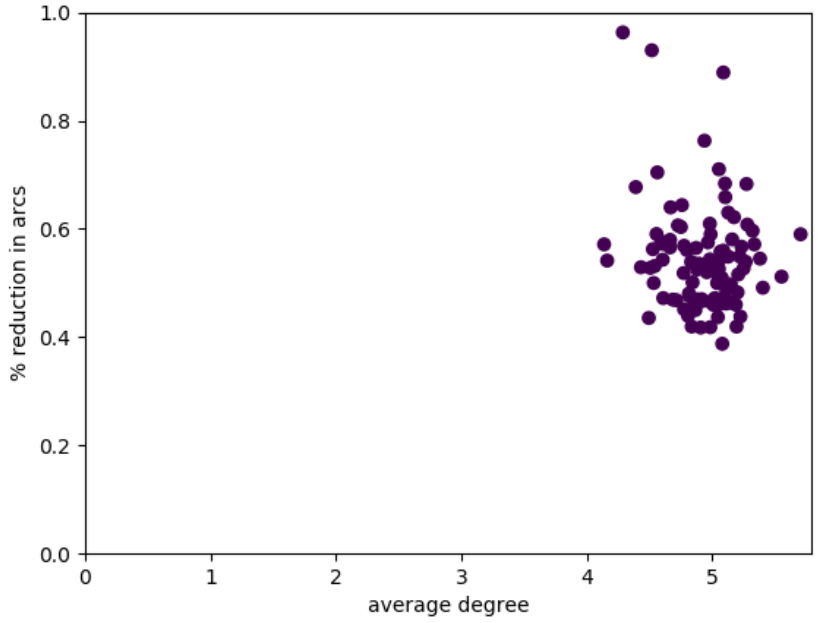
Figure 5.20. Comparison of accuracy for random typing graphs

The RTG reduced node count is roughly 1/3 of the original graphs, with a few outliers, and the edges are reduced by half. This is a better reduction than the random bipartite graphs, and similar to the best cases of the RGGs. However, the RGGs showed significantly more variability in the reduction. Apart from three outliers, the RTGs were all reduced to half or less of the original size, with low variability. The variation for this class of graph does not seem to be correlated with the average degree.

This class of graph shows the lowest error of the classes in this paper, indicating that this coarsening heuristic is reasonable for random typing graphs, representative of large real world networks.



(a) pct node reduction



(b) pct edge reduction

Figure 5.21. Reduction in nodes and edges for random typing graphs

5.6. Conclusion and Future Work

In this chapter, we examined the impact on the Maximum Concurrent Flow Problem of coarsening several classes of graphs using a Maximum Adjacency Search to find communities. The random bipartite graphs showed the most predictable behavior, though lacking significant community structure. The reduction and errors were easily correlated with the minimum degree of the graph. While random typing graphs, with the most modularity, showed the least variability in results and the best improvement in graph size. In most cases, we saw runtime of 1/10 and memory utilization of 1/2 of the original graphs, due to needing roughly 1/5 to 1/2 of the original nodes and 1/2 to 90% of the original edges. The Maximum Adjacency Search with community detection based on edge cut weight fluctuations appears to be a fast method of coarsening graphs, though subject to some small error. We proved that a first cut without ties must be triangle-free.

We also used coarsening of the HMCFP and divisive MCFP solutions to represent clustering to show cluster interactions not captured by the dendrogram. This representation reveals at a glance the clustering improvements offered by the HMCFP over the divisive method previously employed.

Future work includes additional investigation into random typing graphs with different generation parameters beyond the ones used here and varying the community detection method within the maximum adjacency search. For instance, using the level function at node removal or the backwards pass used for partitioning by edge connectivity.

Chapter 6

BOUNDS ON MAXIMUM CONCURRENT FLOW IN RANDOM BIPARTITE GRAPHS

This article originally appeared in [121], and is presented here with some minor modification and the citation in compliance with their copyright policy where the author may reuse the data in their thesis or dissertation.

6.1. Abstract

Algorithms that exploit the duality between maximum concurrent flow and sparse cuts in a graph can effectively discover hierarchical community structures in social networks. We analyze the maximum concurrent flow on random bipartite graphs. Two bounds are discussed based on graph structure. One is the minimum degree bound, based on a sparse cut comprising the edges incident to a node of minimum degree. The other is the distance bound, which occurs when all edges are saturated with flow. We find that the distance bound is constraining when the minimum degree of the graph is sufficiently large, and provide a way to calculate this bound on graphs of diameter three, which occurs with high probability on larger graphs.

6.2. Introduction

We use this the triples formulation in the experiments described in this chapter to improve utilization of computing resources. Along with the memory utilization improvements, this formulation is easy to imagine in dense random bipartite graphs for distance 2 and distance 3 in graphs where the D_3 heuristic is appropriate.

While the triples formulation affords significant memory savings compared to the node-edge formulation, it still requires over 1GB to solve relatively small graphs (e.g. a 128 node random bipartite graph with edge probability of 0.8 or higher). This also takes several

minutes on modern commodity hardware. In this chapter, we explore efficiently calculated bounds on the maximum concurrent flow on random bipartite graphs. In particular, we evaluate the minimum degree bound and the shortest path bound, calculable in constant time and $O(|V|^3)$ time respectively. Additionally, we propose a new heuristic, the D_3 bound, which estimates the shortest path bound assuming that the graph has diameter three and is gridlocked. For graphs of diameter three or less, the D_3 bound is exact. Later discussion will show that this bound can be calculated in constant time from $|E|$ and $|V|$ on random bipartite graphs, which have diameter three with high probability as the size of the graph increases.

The remainder of the chapter is organized as follows: Section 6.3 shows the difference between the bounds we are analyzing on maximum concurrent flow compared to previous results. Section 6.4 shows how to determine critical edges for maximum concurrent flow and covers several observations about them. Section 6.5 explains the shortest path bound using the concept of a fully flow critical graph and derives the D_3 bound. Section 6.6 explores experimental results on over 6,000 random bipartite graphs for the bounds and heuristics compared to the maximum concurrent flow on those same graphs.

6.3. Comparison to Other Gap Analysis Studies

There have been numerous studies of the gap between the MCF and the sparsest cut, beginning with Shahrokhi and Matula [112] finding a $O(\log |V|)$ gap, concurrently found by Leighton and Rao [81]. This was later refined by Arora, Rao, and Vazirani to be $O(\sqrt{\log |V|})$ using expander flows and semidefinite programming [11]. Chuzhoy and Khanna show that gap is $\tilde{\Omega}(|V|^{\frac{1}{7}})$ for multicuts in directed graphs [34]. In contrast to these gap analysis studies, we attempt to approximate the MCF using heuristics and empirically analyze the gap between these heuristics and the true MCF.

Bonsma shows in [23] that the sparsest cut in a complete bipartite graph $K_{m,n}$ with $m \leq n$ and $n \geq 2$ is $min\left\{\frac{1}{2}, \frac{m}{n+m-1}\right\}$, which asymptotically approaches $\frac{1}{2}$ as the size of the graph grows. However, we focus on the MCF, which is less than the sparsest cut in certain

cases. We further expand this thought, by tackling random bipartite graphs where edges have been removed with constant probability and formulating a method of approximating the maximum concurrent flow in constant time.

6.4. Finding Critical Edge Sets For Maximum Concurrent Flow

The throughput of a feasible concurrent flow can grow until a set of edges are saturated sufficiently to prevent a concurrent increase in flow between all pairs. It is useful to review some of these results [6,19,33,41,81,92,94,112] in order to guide our investigation of maximum concurrent flow in bipartite graphs.

An edge of $G(V, E)$ is termed *critical* for concurrent flow if it is saturated by every maximum concurrent flow, or termed *slack* otherwise. The following few paragraphs present some observations on properties of critical edges. Some have been presented before, and are provided for convenience, while others are new contributions of this chapter. The justification section of each indicates whether the observation is a new contribution.

Observation 1. *There exists an optimal flow function $f^*(p)$ on the paths of $G(V, E)$:*

(i) $f^*(p)$ saturates only the critical edges with all slack edges having excess capacity,

(i) the critical edges partition the graph into k components $A_1, A_2, ..., A_k$ for some $2 \leq k \leq |V|$ uniquely determined by the graph, with all edges between A_i and A_j for $1 \leq i < j \leq k$ saturated by $f^*(p)$,

(i) for every $1 \leq i \leq k$, all edges between pairs of nodes of A_i have excess capacity for flow $f^*(p)$.

Justification: For each edge $e \in E$, either e is critical or there is a maximum concurrent flow function $f_e(p)$ that leaves excess capacity on edge e. Then letting $f^*(p)$ be a linear combination of maximum concurrent flow functions for all $e \in E$ for which such flows exist leaving excess capacity in e, we obtain a flow function which is also a maximum concurrent flow that saturates only the critical edges justifying (i).

The slack edges cannot span the graph G since otherwise we could increase the flow between all pairs, so the slack edges must span k components A_i, $2 \leq i \leq |V|$, forming a uniquely determined partition of $|V|$ which could be $|V|$ parts when all edges are critical.

Furthermore, no edge between nodes of any A_i can be critical because otherwise some flow could be rerouted between these nodes on the spanning slack edges of A_i, justifying (iii).

This was previously covered in detail in [93, 112]. □

For our purposes here we shall be interested in the maximum concurrent flow reaching the tightest of two straightforward bounds. One of the bounds derives from the density of cut sets of edges and the other from an array of shortest path distances. Let $(A, \overline{A})$ denote the set of edges of the cut separating $A \subset V$ from $\overline{A} = V - A$.

Observation 2. *Let z^* denote the maximum throughput of concurrent flow in $G(V, E)$, then $z^* \leq min \frac{|(A,\overline{A})|}{|A||\overline{A}|}, A \subset V, 1 \leq |A| \leq |V| - 1$.*

Justification: For any cut $(A, \overline{A})$, a maximum concurrent flow function $f^*(p)$ must have flow of z^* between every pair $v_i \in A, v_j \in \overline{A}$, and in total at least $z^*|A||\overline{A}$ flow must pass through the $|(A, \overline{A})|$ unit capacity edges of the cut $(A, \overline{A})$.

The number of edges $|(A, \overline{A})|$ of a cut divided by the maximum number possible $|(A||\overline{A})|$ for this cut measures the density of a cut and a cut of minimum density is termed a sparsest cut. We shall refer to the bound of Observation 2 as the sparsest cut bound, and if the cuts are limited to removing a single node of minimum degree we refer to the upper limit in Observation 2 as the minimum degree concurrent flow bound.

This was previously covered in detail in [19, 93, 98, 112]. □

Observation 3. *If the critical edges partition the graph into k parts with $2 \leq k \leq 4$, then the sparsest cut bound is tight.*

Justification: In [93], possible graph classes are enumerated for sizes 2, 3, and 4. These are extensible into k-partition archetypes. The chapter also shows that for those classes, the sparsest cut to maximum concurrent flow duality bound is tight. Further, it shows

that the bipartite graph $K_{3,2}$ provides the smallest example where the sparsest cut density (1/2) is greater than the maximum throughput. The maximum concurrent flow for $K_{3,2}$ is $z^* = 3/7$, with the flow $f^*(p)$ determined by splitting the flow equally on all paths of length two between each of the non adjacent pairs. This observation is also used in [112]. □

Note that if there are just two parts then the concurrent flow can be limited (by rerouting if necessary) to paths that cross the cut at most once. If there are three or four parts, then these parts can be similarly shown to identify two or more sparsest cuts that together result in the three or four part partition.

The result for $K_{3,2}$ is an example of the following *shortest path bound* on maximum concurrent flow. It provides the smallest example where the sparsest cut, here being 0.5, is not tight. The maximum concurrent flow for $K_{3,2}$ is $z^* = 3/7$, with the flow $f^*(p)$ determined by splitting the flow equally on all paths of length two between each of the non adjacent pairs.

Observation 4. *The maximum throughput of concurrent flow in* $G(V, E)$ *is* $z^* \leq \frac{|E|}{\sum_{i<j} dist_{ij}}$ *where* $dist_{ij}$ *is the length (edge count) of a shortest path from* i *to* j*. This bound is tight at equality if all flow is on shortest paths and all capacity is utilized (i.e., all edges are critical).*

Justification: Any flow from v_i to v_j must travel through at least d_{ij} edges and use $z^* d_{ij}$ units of capacity of those edges. The capacity utilized by all concurrent flow is then at least $z^* \sum_{i<j} d_{ij}$ which must be less than the total capacity $|E|$. The bound is tight providing equality if all flow is on shortest paths and all capacity is utilized, where then all edges have no excess capacity and all edges are critical. This topic was covered in [96] and [93], which introduced the term *edge-path-regular*. Graphs of this type have the property that the above shortest path bound results in equality rather than inequality. □

The shortest path bound for $K_{3,2}$ is then $z \leq \frac{6}{6\times1+4\times2} = \frac{3}{7}$, as realized by the flow function previously described.

A graph where all edges are critical for a maximum concurrent flow is termed *fully-flow critical* for concurrent flow, and Observation 4 provides a tool for verifying that a graph is fully-flow critical.

We refer to the bound given by $\frac{|E|}{\sum_{i<j} \text{dist}_{ij}}$ as the *shortest path bound* in the general case. Since this calculation of the shortest path bound does not reveal which edges are critical, it cannot be checked to see if the bound is tight against the maximum concurrent flow without solving the LP. This term is used rather than *gridlock bound* when evaluating heuristics later in this chapter. The *gridlock* condition occurs when the shortest path bound is tight and is a synonym for *fully-flow critical* derived from traffic analogies for the maximum concurrent flow problem. Recently, [99] found that most graphs that gridlocked in their experiments on random graphs were of diameter two, with a possibility of gridlock at diameter three. Random bipartite graphs have a low diameter, but with structure that may gridlock more often even with a higher diameter than two or three.

For graphs possessing considerable symmetry such as complete bipartite graphs, the following result provides identification of a large class of graphs that must be fully flow critical. A graph is termed *edge-transitive* if for every pair of edges $e_1, e_2 \in E$ of the graph there is an automorphism of the graph mapping edge e_1 into e_2.

Theorem 6.1. *Every edge-transitive graph is fully flow critical.*

Proof: Suppose $G(V, E)$ is edge-transitive with automorphism $\phi(V)$, and $f_{e1}(p)$ is a maximum concurrent flow function where edge $e_1 \in E$ is not critical. Then for any other edge e_2 there is an automorphism mapping e_1 into e_2 . This automorphism maps paths of G into corresponding paths of G employing edges of $\phi(V)$. These paths may be assigned the flows from $f_{e1}(p)$ and we obtain a maximum concurrent flow of G termed $f_{e2}(p)$ where e_2 must also be slack. Since e_2 is any edge of G, and since a linear combination of maximum concurrent flow functions is a maximum concurrent flow $f^*(p)$ that would have all edges slack, this contradiction verifies that all edges are critical. □

There are many known classes of edge-transitive graphs such as all complete bipartite graphs $K_{m,n}$ and $K_{n,n} - M$ where M is a perfect matching. While there are many edge-transitive classes other than bipartite graphs such as the Petersen graph and complete multipartite graphs where each part has the same size, we focus on bipartite graphs.

Following the widely studied evolution of random graphs introduced by Erdös and Rényi [45] and summarized in several texts we shall investigate the maximum concurrent flow in random bipartite graphs, particularly for the subgraphs of $K_{n,n}$.

A *random bipartite graph* $G(n, n, p)$ denotes a subgraph of $K_{n,n}$ selected by deleting each edge of $K_{n,n}$ independently with probability $(1 - p)$, that is, each edge of $K_{n,n}$ is retained with probability p. We shall treat evolution in the sense of randomly removing edges from $K_{n,n}$ as expressed by investigating properties of $G(n, n, p)$ as p decreases.

In our results, over 78% of the random bipartite graphs are fully flow critical. We shall provide more detailed results indicating that most random bipartite graphs are fully critical in the sense that for any fixed p and sufficiently large n, it is highly likely that the graph $G(n, n, p)$ is fully flow critical.

An easily tested condition for a bipartite graph to not be fully flow critical is provided by the following.

Observation 5. *If the random bipartite graph $G(n, n, p)$ has a minimum degree δ such that $\delta/(2n - 1)$ is strictly less than the shortest path bound given by $(|E|/\sum_{i<j} dist_{ij})$ then the graph is not fully flow critical.*

Justification: A fully flow critical graph satisfies the equality constraint $z^* = \frac{|E|}{\sum_{i<j} \text{dist}_{ij}}$. If the minimum degree bound $\delta/(2n - 1)$ is less than the shortest path bound, then the shortest path bound is not tight. Therefore the graph cannot be fully flow critical. □

Observation 6. *A random bipartite graph $G(n, m, p)$ is more likely to be a complete bipartite graph minus a matching ($K_{n,m} - M$) as p increases.*

Justification: The probability that the edges removed from $K_{n,m}$ (the bipartite complement) form a matching can be expressed as a modification of the hypergeometric distribution. Let $e' = \lfloor n * m * (1 - b) \rfloor$ be the number of missing edges where b is the bipartite density, or the proportion of edges in the graph compared to $K_{n,m}$. If e' represents the number of samples, then the maximum number of missing edges on a given node i in n is $min(e', m)$.

Then the probability that k missing edges are incident upon i can be given by

$$P(X = k) = \frac{\binom{m}{k}\binom{n*m-m}{e'-k}}{\binom{n*m}{e'}}$$

Extending this to the condition of $P(X \geq k)$ and iterating over each node in n and m, the full equation becomes

$$P(X \geq k) = |m| \sum_{i=k}^{min(e',m)} \frac{\binom{m}{k}\binom{n*m-m}{e'-k}}{\binom{n*m}{e'}} + |n| \sum_{i=k}^{min(e',n)} \frac{\binom{n}{k}\binom{n*m-n}{e'-k}}{\binom{n*m}{e'}}$$

By setting $k = 2$, then $1 - P(X \geq 2)$ becomes the probability that $G(n, m, p)$ is an instantiation of $K_{n,m} - M$, and therefore gridlocked.

For some fixed k, if p approaches 1, then b approaches 1, and e' approaches 0. It is easy to see that $\lim_{e' \to 0} P(X \geq k) = 0$, so the probability that $G(n, m, p)$ is $K_{n,m} - M$ increases monotonically to 1. □

In our computational results determining the maximum concurrent flow throughput, we explicitly cite those cases that are fully flow critical and those throughputs constrained by a minimum degree sparsest cut bound.

6.5. Fully Critical Flow For Bipartite Graphs Of Diameter Three

There are several properties of bipartite graphs in general and for random bipartite graphs specifically that suggest that most bipartite graphs are fully flow critical. An important step is justifying that most bipartite graphs have diameter three.

Lemma 6.2. *Any induced subgraph of $K_{n,n}$ with minimum degree $\geq (\frac{n+1}{2})$ must have diameter three.*

Proof: With $K_{n,n}$ having partitions A and B, any two points of A must have at least $(n + 1)$ edges to B and therefore have at least one path of distance two between them. Consider $a \in A, b \in B$, with a not adjacent to b, and $c \in B$ be adjacent to a. Now c has at least one path of distance two to b which can not include a since a is not adjacent to b.

Therefore any non-adjacent node pair from A and B must have distance three. □

Lemma 6.2 implies that if we remove edges from $K_{n,n}$ in any order, particularly any reasonably uniform order, we must remove a large number to increase the diameter above three. Bollobas and Klee [22] have given many threshold results for diameters of random bipartite graphs including that for any p and sufficiently large n, the diameter of $G(n, n, p)$ is almost surely three.

For computational purposes we can employ the following elementary bound for the expected number of node pairs with distance three.

Observation 7. *The expected number of node pairs* $\{i, j\} \in G(n, n, p)$ *with even* $dist_{ij} \geq 4$ *(i.e.,* $dist_{ij} = 2 + 2t$ *for* $t = 1, 2, \cdots$*) is* $q = n(n-1)(1-p^2)^n$.

Justification: Consider node pairs $\{a_1, a_2\}$ from one side of the bipartition and $\{b_1, b_2\}$ from the other. The probability that both (a_1, b_1) and (a_2, b_1) are edges in E is p^2. Thus, the probability that a_1, b_1, a_2 is *not* a path in $G(n, n, p)$ is $(1-p^2)$. Likewise, the probability that the path a_1, b_2, a_2 does not exist is $(1-p^2)$. Since the events that paths a_1, b_1, a_2 and a_1, b_2, a_2 exist are independent, The probability that there is no path of length two between a_1 and a_2 is $(1-p^2)^n$. The observation follows from the fact that there are $\frac{n(n-1)}{2}$ node pairs on each side of the bipartition. □

Setting $n = 50, q = 0.1$, and solving for p yields $p = 0.4278$. Thus, the expected number of node pairs $\{i, j\} \in G(50, 50, 0.43)$, with distance greater than 3 is on average less than $q = 0.1$, which implies that the event of diameter not equal to three being less than 0.1 occurs for smaller p, specifically the probability that the diameter of $G(100, 100, 0.3296)$ is greater than three is less than 0.1. Alternatively we see that for larger n and fixed p the probability for a diameter greater than three becomes progressively small.

We are particularly interested in bipartite graphs of diameter at most three because the shortest path bound on maximum concurrent flow is then easily computed from the number of nodes and edges of the graph.

Noting that a bipartite graph of diameter three must have $m = |E|$ pairs at distance one, $n(n-1)$ pairs at distance two, and $(n^2 - m)$ at distance three, we obtain $\sum_{i<j} \text{dist}_{ij} =$

$m + 2n(n-1) + 3(n^2 - m) = 5n^2 - 2(m+n)$. Thus we may write the shortest path bound as $D_3 = m/[5n^2 - 2(m+n)]$.

Observation 8. *When the maximum concurrent flow equals the bound* $D_3 = m/[5n^2 - 2(m+n)]$, *then the graph has diameter three and is fully critical.*

6.6. Experimental Results For Random Bipartite Graphs

6.6.1. Problem Statement

Given the analytical results described above, we sought confirmation in experimentation. The experiment also provided additional insights for analysis. Specifically, we evaluated the conditions that allow the degree bound and shortest path bound to match the maximum concurrent flow in random bipartite graphs.

6.6.2. Approach

We created 6,210 random bipartite graphs, $G(2, 2, p)$ to $G(70, 70, p)$, with the same number of nodes in each side of the set bipartition and p ranging from 0.1 to 0.9, inclusive, in increments of 0.1. For each combination of size and probability, 10 graphs were created. A set of 69 similarly sized complete bipartite graphs were also created, for a total of 6,279 graphs. We used the triples formulation of the maximum concurrent flow LP [41] to find the maximum concurrent flow on each graph. We compared these results to the degree bound for a cut that removes the node of minimum degree calculated as $\delta/(2n-1)$, the shortest path bound calculated as $|E|/\sum_{i<j} \text{dist}_{ij}$, and the D_3 bound defined below to evaluate the utility of these bounds as heuristics for the maximum concurrent flow on random bipartite graphs.

The D_3 bound is calculated using the shortest path bound, calculated as though the diameter of the graph were 3. That is, $D_3 = |E|/(p_1 + 2 * p_2 + 3 * p_3)$, where $p_1 = |E|$, $p_2 = 2 * n * (n-1)$, and $p_3 = n * n - |E|$. These values represent the expected number of shortest paths of length 1, 2, and 3 respectively, given the assumption of diameter 3.

6.6.3. Experimental Results

The triples formulation LP of the maximum concurrent flow problem was solvable in seconds on smaller graphs, but took nearly 9 minutes per graph on the $G(50, 50, p)$ graphs and over an hour on the $G(70, 70, p)$ graphs to solve. In the $K_{50,50}$ graph, the constraint matrix was 4,950 rows x 245,001 columns, using 566Mb of memory. While $K_{70,70}$ had a constraint matrix of 9,731 rows x 676,201 columns, using 1549Mb of memory. For reference, $K_{100,100}$ was solved on the same computer, taking almost 14 hours and 4,529Mb of memory.

In contrast, the calculation of the degree bound and D3 heuristic took under a millisecond each. The calculation of the shortest path bound took under two seconds per graph, due to the need to run an all-pairs shortest-path algorithm. This experiment used the Boost Graph Library [113] implementation of the Floyd-Warshall algorithm [48, 123].

Observed Result: The transition from a constraining degree bound to a constraining shortest path bound happens when the minimum degree is roughly $|V|/3$.

From the results, we see that the minimum degree bound is successful in 1,895 of the 6,279 cases. We also see that the shortest path bound dominates the degree bound once the minimum degree is larger than 13. Consider that in order to get to a min degree of 13, the average degree must be at least 13, making the size of the random bipartite graph at least 26. This equates to a degree bound of at most $13/25 = 0.52$. The shortest path bound in this case is at most the D_3 bound of $169/(169 + 156 + 156) = 0.35$. In order for the degree bound to be lower, there must be at least $13/0.35 = 37.14 + 1$ nodes in the graph, or slightly less than $|V|/3$. Figure 6.1 shows this pattern with a color scale where blue is $p(edge) = 0.1$ and red is $p(edge) = 1.0$.

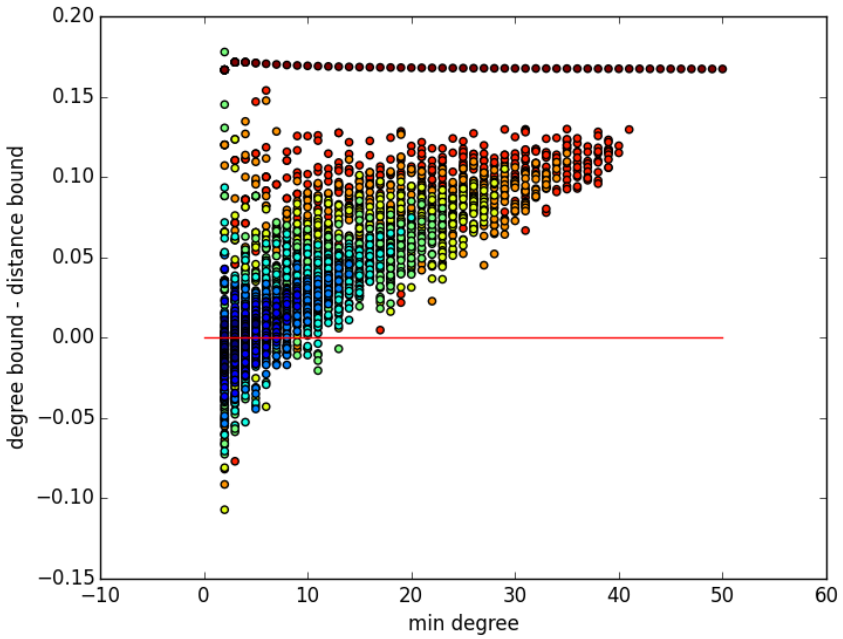


Figure 6.1. Distance Bound Error vs Min Degree

While there is a trend toward smaller errors in larger graphs, there were still errors throughout the sample, as shown in Figure 6.3. We see in Figure 6.4 that the reduction in errors is more strongly a function of the minimum degree than of the size of the graph. We also explored the dependence on $n * p$ with similar findings.

This result appears to be generalizable to $G(n, n, p)$ where $n \geq 13$. Further, we conjecture that it is generalizable to $G(n, m, p)$ according to the following argument. Without loss of generality, assume $|m| \leq |n|$. Then for the minimum degree to be at least 13, $|m| \geq 13$ and it follows that $|n| \geq 13$ with similar results even in cases where $|m| \neq |n|$.

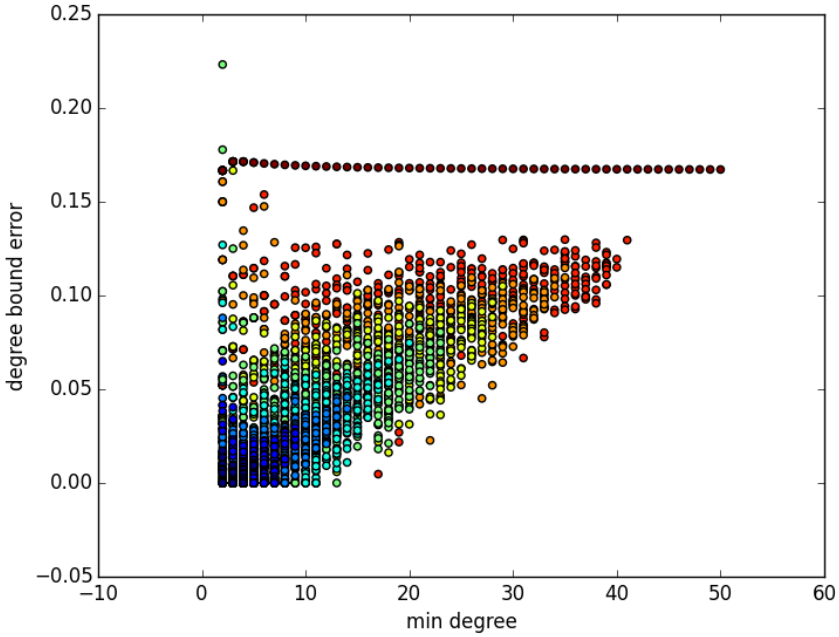


Figure 6.2. Degree Bound vs Min Degree

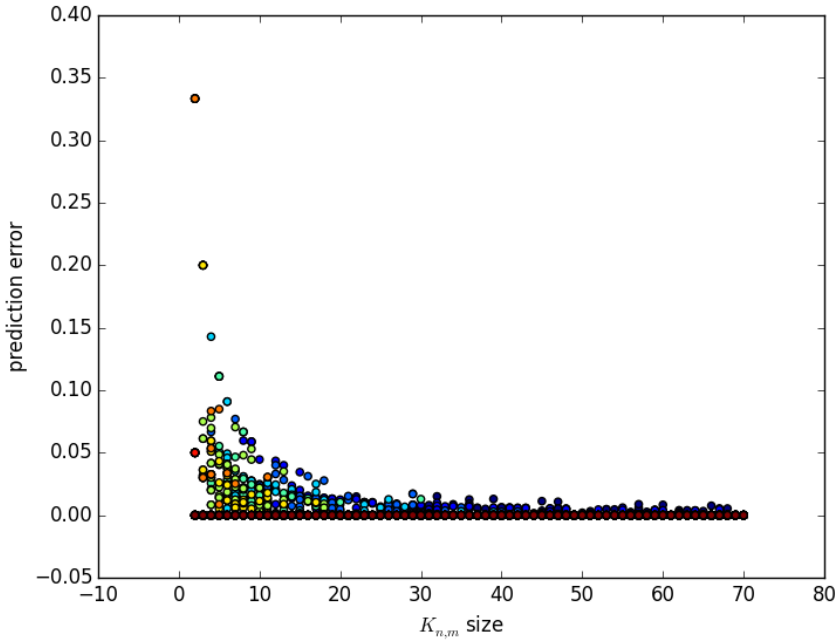


Figure 6.3. Most constraining bound error vs graph size

We also see that the degree bound is not the constraining bound in any graph where the minimum degree was greater than 13 (Figure 6.2). It also does not provide a constraining bound on any of the complete bipartite graphs. However, the distance bound is always constraining for our cases when the minimum degree exceeds 13 and on complete bipartite graphs (Figure 6.5). By taking the minimum of the degree and distance bounds, the new constraint matches the maximum concurrent flow in all cases where the minimum degree exceeds 8 (Figure 6.4).

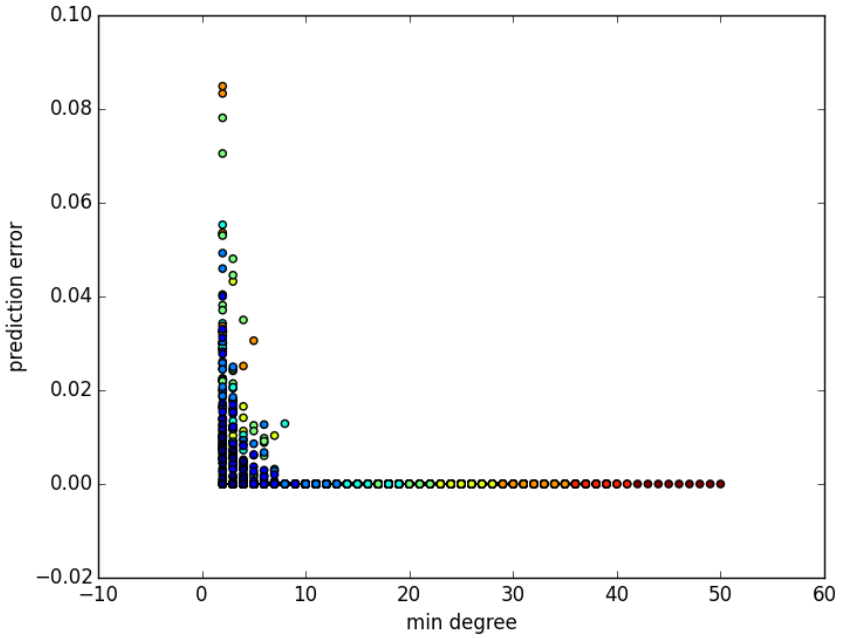


Figure 6.4. Most constraining bound error vs min degree

We show the most constraining bound (i.e. the smaller of the shortest path bound and the degree bound) error versus the graph size and minimum degree. The graph size is given by the number of nodes on one side of the bipartition. The error is calculated as $(bound - z^*)$. Since the optimal flow, z^*, must be less than or equal to the bound, all errors are non-negative.

While there is a trend toward smaller errors in larger graphs, there were still errors throughout the sample, as shown in Figure 6.3. We see in Figure 6.4 that the reduction in

errors is more strongly a function of the minimum degree than of the size of the graph. The dependence on $n * p$ showed similar results.

We also see that the degree bound is not the constraining bound in complete bipartite graph, nor in any graph where the minimum degree was greater than 13. However, the shortest path bound is always constraining for our cases when the minimum degree exceeds 13 and on complete bipartite graphs (Figure 6.5). By taking the minimum of the degree and shortest path bounds, the new constraint matches the maximum concurrent flow in all cases where the minimum degree exceeds 8 (Figure 6.4).

Observed Result: The D_3 bound gives similar results to the shortest path bound.

Bollobas and Klee calculated the expected diameter of a random bipartite graph as a function of n and p [22]. As $n * p$ grows, the probability that the graph will be of diameter two or three increases. The shortest path bound would only deviate from the D_3 bound by adding a small amount to the denominator to represent the paths of length 4 or more, since the number of such paths is expected to be low.

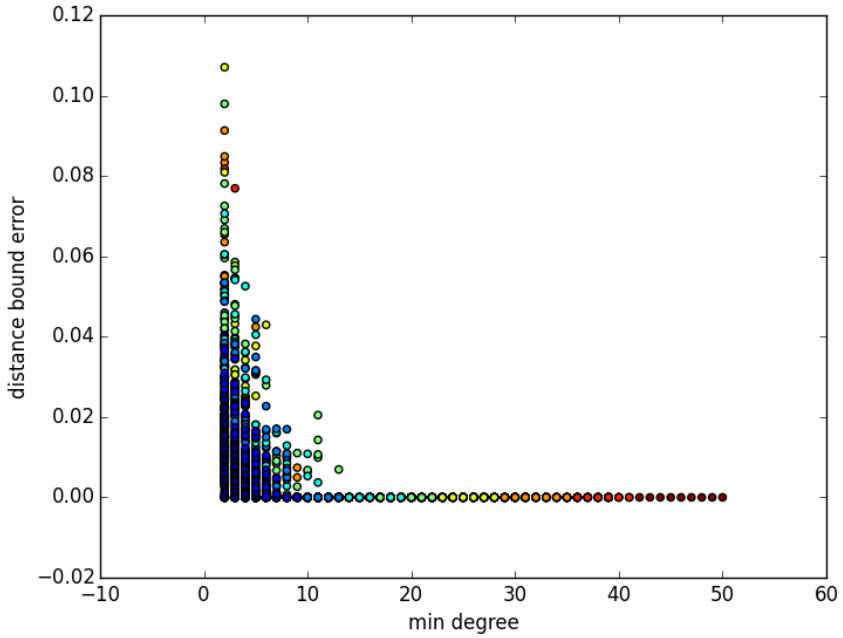


Figure 6.5. Shortest Path Bound vs Min Degree

In our sample, the D_3 bound was limiting in 3,245 / 6,279 (51.5%) cases, and the shortest path bound was limiting in 3,885 / 6,279 (61.8%) cases, or 640 additional cases, with overlap occurring with graphs of diameter three, making the D_3 bound equivalent to the shortest path bound in all the additional cases, $p(edge) \geq 0.5$, decreasing as the number of nodes in the graph increases.

6.6.4. Summary of Experimental Results

Our results show that with high probability, the maximum concurrent flow in a connected random bipartite graph with minimum degree larger than 8 is equal to the shortest path bound. This is true in all cases in our sample when the minimum degree is larger than 13. Therefore, in these cases of a random bipartite graph, the maximum concurrent flow can be well estimated using an all-pairs shortest-path algorithm, rather than a more expensive LP.

We find that the D_3 bound is constraining in most cases, more so as the probability of diameter three or less increases, i.e. as the shortest path bound approaches the D_3 bound. Therefore, if the diameter is known a priori to be two or three, then the maximum concurrent flow is calculable in constant time, without needing all-pairs shortest-path algorithms.

Out of our 6,279 graph sample, we successfully find the maximum concurrent flow in all but 443 cases using a combination of checking for connectedness of the graph, calculating the minimum degree bound, calculating the D_3 bound, and if the diameter is greater than three, the shortest path bound. The remaining cases tended to have a pattern similar to Figure 6.6 where the graph is not an instance of $K_{n,m} - M$, and there are enough nodes of minimum degree that the shortest path bound (0.22) is less than the degree bound ($2/7 = 0.285714$).

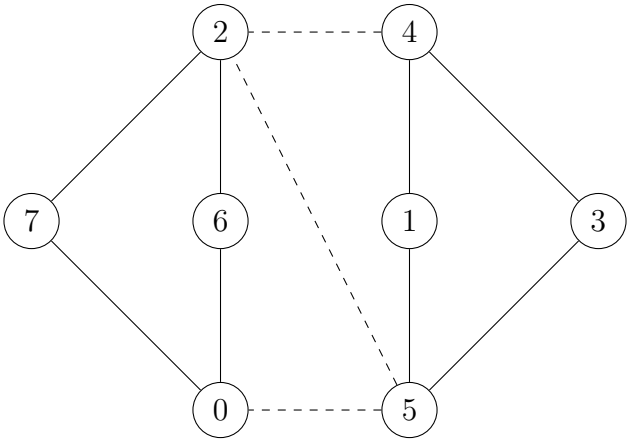


Figure 6.6. Example non-flow critical diameter 3 bipartite graph with saturated edges dashed

6.7. Conclusion

Despite being polynomially bounded, the maximum concurrent flow problem is computationally challenging due to the efficiency of linear programming algorithms and the memory required to construct the problem for calculation. This chapter describes two bounds on the problem. We show that the minimum degree bound is often, but not reliably, constraining on small or sparse random bipartite graphs. However, on random bipartite graphs of larger minimum degree the shortest path bound is nearly always a gridlock constraint, implying that these random bipartite graphs are often fully flow critical.

This chapter also introduces a heuristic for the shortest path bound by assuming that the graph has diameter three and is gridlocked. Random bipartite graphs tend to have diameter three as they grow in size and $p(edge)$, making this assumption reasonable. We find the D_3 heuristic provides a constant time estimate of the maximum concurrent flow in most cases for this class of graph.

Chapter 7

QUANTUM WALKS FOR SPARSE CUTS

7.1. Abstract

We explore the use of three different coins in a discrete time quantum walk to find the sparsest cut in a graph. The fair coin, degree coin, and greedy coin show insignificant differences in performance relative to each other. All of these coins compare unfavorably with classical heuristics in both wall-clock running time and accuracy on several test graphs.

7.2. Introduction

Most research into quantum algorithms has focused on security applications, often related to Shor's factorization algorithm. Other quantum research has dealt with potential improvements in solving NP-complete problems. These have so far only shown improvement for specialized cases of the problems. In this chapter, we focus on a quantum implementation of a random walk for solving the Sparsest Cut Problem. Our heuristic shows poor results compared to heuristics implemented on classical computers, due to slow performance and inaccurate results on test graphs. The approach taken here is an initial approach that checks the value of the coin at each step in the diffusion to update it based on the neighbors. Future versions of the approach could use the entire adjacency matrix to create a coin that does not need an update, thus allowing a full diffusion for a quantum walk.

7.3. Background

The Sparsest Cut Problem is the NP-hard optimization problem of partitioning a graph in such a way that the cut density is minimized [98]. In the 2-partition case, cut density is defined as

$$density = \frac{\sum \text{cut capacities}}{|A| * |A'|} \tag{7.1}$$

The partition is not required to be unique; multiple cuts can tie, resulting in a cut described in literature as k-partite, equipartition, "web-cut", and "gridlock". An in-depth exploration of web cuts is beyond the scope of this chapter, but can be seen in [112]. The problem can also be reduced to an Integer Problem (not linear) by defining the objective function for minimization as

$$\frac{\sum_{i \in E} (x_i * y_i)}{\sum_{j \in PAIRS} (z_i * w_i)} \tag{7.2}$$

where x is an edge capacity and z is a pair of nodes. The variables y and w, respectively, represent binary selection variables to indicate the cut status of an edge and separation of pairs.

The sparsest cut problem has significant application in many fields, though has received little discussion in quantum computing literature to date, even with several of the same people actively publishing in both fields.

Many attempts to solve combinatoric problems on both classical and quantum computers have revolved around graph traversals. In general, a value, here the cut density, is calculated at each step during a search and the best answer tracked. For NP problems, an oracle may be needed to break ties correctly so that the correct answer can be found.

Quantum computing offers the possibility of just such an oracle for traversals in the form of a quantum walk with a coin [114]. Quantum walks on graphs were traditionally used to find a particular node or edge, rather than a graph partition. This approach has been shown to provide good results for polynomially bounded problems such as the minimum spanning tree, graph connectivity, and single source shortest path problems in [43]. The choices need stochastic probabilities assigned at each stage so that a coin can be flipped to determine the correct path. Grover's algorithm for unstructured search [58] can be viewed as a quantum walk on a complete graph to find a marked node [114], earning the name "Grover coins" for the technique.

Quantum walks were originally implemented using a continuous time diffusion where the Hamiltonian is given by the adjacency matrix for the graph. Then the graph state (position on the graph) is given by $U = exp(-i * H * t)$ [74]. For many people familiar with graph operations, a discrete time walk is more intuitive, though it was difficult to implement for some time [7]. In a discrete time walk, the diffusion is given by $|x_{i+1}\rangle = S * C * |x_i\rangle$, where S and C are the switch and coin operators respectively. Some papers have also written this as $|x_{i+1}\rangle = S * (C \otimes I_v) * |x_i\rangle$, to explicitly state the expansion of the coin into the full state, where I_v is the identity matrix of size $|V|$ [1]. The first implementations were on d-regular graphs where a non-regular graph gets self-edges (loopback) added to every node with less than maximal degree [74]. The probability of remaining at that node at each stem is then represented by the proportion of self-edges to other edges.

Compared to an exponential speedup in other cases, Grover's algorithm only offers a $\sqrt{(N)}$ speedup. This is still relevant in NP problems because they are solved exactly for sufficiently small cases. A reduction of 2^N to $2^{N/2}$ increases the size of the largest tractable problem. The only two exact algorithms found for the sparsest cut problem are the naive on of $O(2^{|V|})$ checking everything and the non-linear integer programming problem in [57]. It is hoped that an exploration of the field using a quantum walk can improve on these without requiring a full solution to create the coin for discrete walks or the Hamiltonian for continuous time walks.

7.4. Approach

Our approach is to traverse the graph using a quantum walk, while tracking the least cut density found. We impose a stopping condition when all nodes have been visited at least once. Due to the random nature of the walk, a node may be removed from A and revisited several times. Therefore, the walk is not guaranteed to terminate without an additional condition. We limit the number of iterations to be $2 * |V|^3$, which may not be enough to find the correct answer. In order to find the correct answer, an exponential number of steps may be needed for NP-complete problems such as this one [114].

Similar to [8], we represent state space as $|d, v\rangle$. In our case, v is still the number of qubits to hold all nodes of the graph, given as $\lceil log|V| \rceil$. However d is enough qubits to hold the max degree of the graph, $\lceil log\Delta \rceil$, which in the worst case is $\lceil log(|V| - 1) \rceil$.

Each coin ket was converted to a diffusion operator in the traditional way of

$$C = 2 |c\rangle \langle c| - I_d \tag{7.3}$$

For non-power-of-2 sizes, this results in a matrix of [7]

$$C = \begin{pmatrix} \boldsymbol{C}_{deg(i)} & \mathbf{0} \\ \mathbf{0} & -I_{\Delta - deg(i)} \end{pmatrix} \tag{7.4}$$

The first bits in the state are used for the coin operator. In our experiment, we designed 3 coins to govern the walk. The first of which is a fair coin, using the Grover diffusion operator. This coin has been commonly used in discrete quantum walks on graphs [7]. To create this, we created a ket of d qubits for each target node, and summed all the kets. The sum is at least 1, so the result needed to be normalized to adhere to the normal superposition rules for probability of being in each state.

$$|c\rangle = \frac{1}{\sqrt{deg(v)}} \sum_{e \text{ adj } v} |e\rangle \tag{7.5}$$

Transforming the ket into a diffusion operator gives a d x d matrix, shown below where $k = deg(v) < d$ to illustrate how the kets fill the coin space in that condition

$$C = \begin{pmatrix} -1+\frac{2}{k} & \frac{2}{k} & \cdots & \frac{2}{k} & 0 & \cdots & 0 \\ \frac{2}{k} & -1+\frac{2}{k} & \cdots & \frac{2}{k} & 0 & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots & \cdots & \cdots & \cdots \\ \frac{2}{k} & \frac{2}{k} & \cdots & -1+\frac{2}{k} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & -1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & -1 \end{pmatrix} \tag{7.6}$$

When the node has only 2 adjacent edges, the useful part of the fair coin for the current node reduces to the Pauli-x matrix [7]: $\sigma_x = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$.

Second, we biased the coin using the degree of the target node. Where in the previous coin, the kets were summed with unit value each, in this case, the ket has a factor of the degree. This is the normalized as above. We will call this the "degree coin" in the remainder of the paper.

$$|c\rangle = \frac{1}{\sqrt{\sum_{\text{eadj}v} deg(e)}} \sum_{e \text{ adj } v} deg(e) \, |e\rangle \tag{7.7}$$

Finally, we created a coin that favored decreasing cut density by evaluating the result when each target node had its parity flipped individually. These densities were then summed and the difference from the sum was used as the value for each target ket. The resulting ket was then normalized as above to reflect proper superposition. This is approximately a greedy approach, with some probability of not taking the "best" answer, and will be called the "greedy coin" in the rest of the paper.

$$|c\rangle = \frac{1}{\sqrt{\sum_{\text{eadj}v} value(e)}} \sum_{e \text{ adj } v} value(e) \, |e\rangle \tag{7.8}$$

The switch operator is somewhat simpler than the coin, being the sum of all projections along adjacent edges, controlled by the coin. $S\left|d,v\right\rangle = \left|d,v'\right\rangle$ and $S\left|d,v'\right\rangle = S\left|d,v\right\rangle$ [73]. This is given by

$$S = \sum_{i=1}^{deg(v)} \left(\left|i\right\rangle\left\langle i\right| \otimes \left|v'\right\rangle\left\langle v\right|\right) \tag{7.9}$$

In order to make decisions at each node, a measurement is taken before traversing an edge. This reduces the quantum walk to a classical random walk, still maintaining the log-space advantages of a quantum computer.

7.5. Results

We implemented each of the 3 coins mentioned above using the QuTIP library [66, 67]. We then tested the performance on each of the 8 graphs used in [19], labeled in the figures below as g1-g8. The walk was repeated from the beginning 20 times, to get several samples of each walk for comparison. Simulations of quantum algorithms on classical computers are notoriously slow, and this case is no different. It took 18:02:47, 20:50:12, and 17:02:36 for the fair coin, degree coin, and greedy coin, respectively.

Figure 7.1 shows that once the correct answers are removed from the set of results, the average error does not vary significantly by coin type.

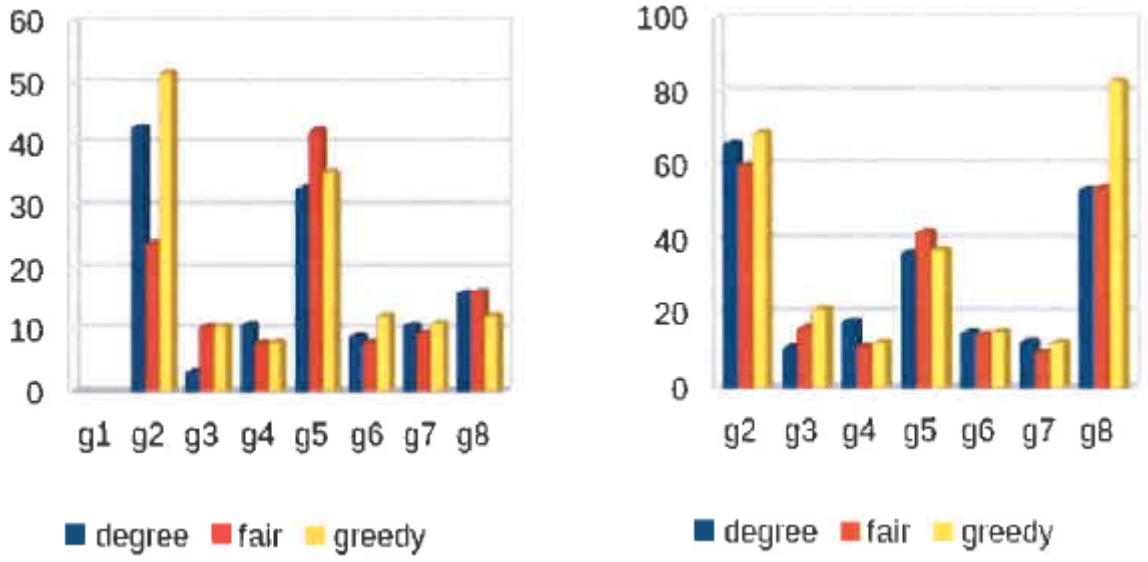


Figure 7.1. Mean % error (a) over all cases and (b) cases with incorrect results

In general, the degree coin got the right answer more often than the greedy coin, as shown in Figure 7.2. The only exception is graph 8, which was specially constructed to be easy to solve for classical computing algorithms.

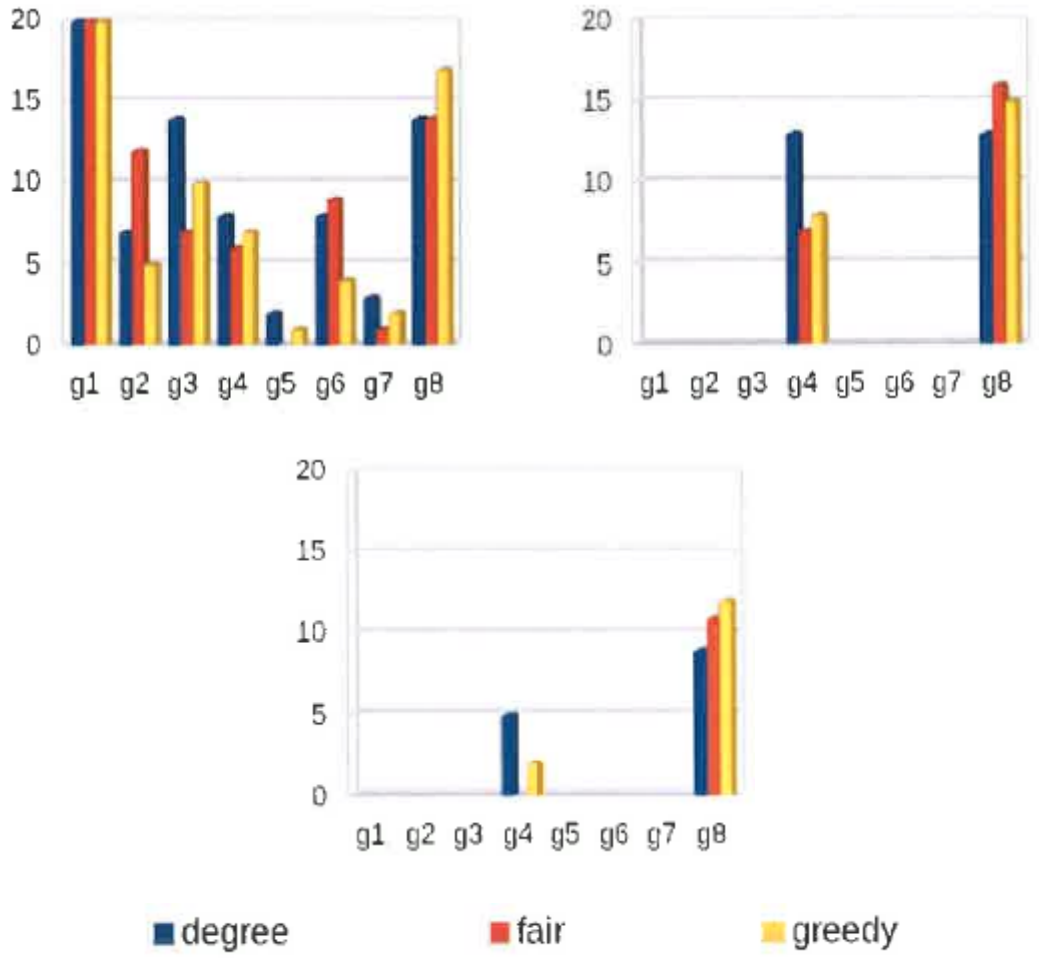


Figure 7.2. Counts of (a) correct answers, (b) termination due to iteration limit, and (c) terminations due to iteration limit with correct answers, by coin type.

Only graphs 4 and 8 saw the algorithm terminate due to max iterations. Most of those cases in graph 8 also fond the correct answer, while this was not the case for graph 4. In fact, when the fair coin took the full set of iterations, it did not find a single correct answer on that graph.

7.6. Discussion

The fair coin was designed to be a basis for comparison against the other two coins. However, the success rate was higher for different coins on different graphs. Graph 8 was the

only case where the greedy coin found the most correct answers (not tied with other coins). The degree coin found the most correct answers without ties on four graphs, the most for any coin. When the correct answers were removed, there was no clear winner by measuring the size of the average errors compared to the right asanswer

Our example coins did not show a significant and repeatable benefit over the fair coin. As mentioned above, this may indicate that more reps are needed in each case. It is more likely that either different coins are needed, or the diffusion model modified to be measured less frequently (an actual quantum diffusion). These coins were not biased against backtracking, which may be an avenue for exploration as well.

The use of the 8 sample graphs helped check the coins by providing a diverse set of graphs for analysis. The sparsest cut problem has so far been very difficult on large graphs, and quantum simulations on a classical computer would only make the time constraints worse.

Most quantum walks take advantage of superposition to reduce the measurement frequency. This also allows each node to diffuse in parallel. As our walk is measured after each coin flip, we were unable to explore those advantages here. This reduces the algorithm to a classical random walk implemented on a quantum computer, providing only the log-space advantages of superposition.

One heuristic on a classical computer under investigation by the authors is extending the min-cut algorithm from Stoer and Wagner [117] to find sparse cuts. For all eight sample graphs, this algorithm using Maximum Adjacency Search [95] found the right answer, though it fails on other some graphs. A longer discussion on the MAS is available in Chapter 4 These other graphs have at least 200 nodes, and become intractable for classical simulations of quantum computers.

7.7. Conclusion

Our work showed that a coin biased toward nodes of high degree and a coin biased toward smaller cut densities do not offer significant improvement over a fail coin when measured at each step. We covered a brief background on the sparsest cut problem, then explored

the background for developing these coins, derived from multiple sources. The long running time of the simulations prevented a larger investigation of the problem, but initial results indicate the need for a different coin or different algorithm.

Future work may include attempts to further refine the coin(s) and measuring less often. It may also be of interest to allow each node selected to choose an edge at each iteration, possibly one already used. This could resolve the inaccuracies seen on some graphs above. Additionally, it contributes to reducing the measuring frequency by configuring the code to allow for the diffusion over multiple steps. In our experiment, the walk started at the first node, due to some arbitrary ordering, each time. Some dependence on the starting node has been seen in previous work relating to cuts such as [117]. Therefore, future work could also include varying the starting node to search for impacts there.

Chapter 8

COMPUTING ENVIRONMENT

The architecture, libraries and design of the software allow easy experimentation with algorithms and data sources to provide a robust solution.

In order to ensure consistency across all runs, especially for the timing studies executed, we ran all cases on the same laptop computer with 8Gb of RAM, 24 Gb of swap space, with an Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz. The laptop was running Slackware Linux 14.2 with GLPK 4.63 as the LP solver.

8.1. Libraries

The software was written in C++17 with some processing scripts written in Python 3 for data analysis, including visualization of statistical results.

We used the GNU Linear Programming Kit (GLPK) as the primary LP solver. Our initial investigation into which library to use examined CPLEX, Gurobi, LPSolve, and GLPK. Both CPLEX and Gurobi were initially ruled out due to their closed source nature, even though they have been shown to perform faster on this class of problem. We also ruled out LPSolve [17] when it took significantly longer than GLPK to solve some small problems. GLPK provided an easy to use interface and reasonable, though not the best, performance while still being open source.

We also used the Boost Graph Library (BGL) [113] for its robust suite of graph algorithm implementations. Initial evaluations of alternatives included the KDE Rocs suite, and multiple hand-rolled solutions some used homemade data structures and others using the Eigen3 library [59]. The Eigen3 library provided significant speedup at the expense of contiguous memory allocation even for large graphs, where the BGL offered flexibility and some algorithms already implemented. After choosing to work in the BGL, we contributed updates to

the library to expose the MAS algorithm from where it was previously inaccessible, buried in the Stoer-Wagner algorithm.

For numerous C++ convenience functions and a user interface (UI) framework, we used the Qt library. This helped provide an experimentation environment with a single UI for development and multiple command line executables for batch functions of single algorithms for both graph generation and evaluation.

Of the different heuristic algorithms covered, most were developed in C++. Only the quantum coins were developed in Python3 using the QuTIP library [66,67]. This library was developed to provide multiple methods of developing quantum computer simulations. Most of the work has focused on simulating qubits themselves, like ion traps. Though some of the library supports quantum algorithm development. QuTIP supports the quantum annealing method used by popular quantum computers like those from D-Wave. It also supports the Deutcsh gate model used in Chapter 7.

Finally, we used Matplotlib [65] for generation of analysis graphics after reading the results files into NumPy [103] arrays. Matplotlib provides an API similar to the figure API in MatLab but in Python3, for use in similar visualization needs. We used it in multiple chapters in this paper to create any figures not produces in our Qt-based UI or tikz in LaTeX.

8.2. Architecture

All of this was brought together in a single GUI built in Qt for visualization of results on a single graph and to rapidly switch between graphs and the algorithms used on the current graph. Keeping the UI layer separate from the data processing layer pushed the architecture to event driven design, especially given the long runtimes of the cut finders and the need for user interaction. A graph factory was developed to create hard-coded graphs like the ones from the Biswas paper or the NCAA graph, and to read in files on disk in a variety of formats, including GLPK and some of the SNAP formats.

The cut finders were developed from a base class, so that they could be used as part of a strategy pattern for the UI and in the command line runners for batch processing of graphs.

Most of the functions are in the base class, then each implementation class used the *findCut()* method to implement their specific algorithm. As seen in Figure 8.1, we implemented the MAS heuristic and the LP using GLPK. These were combined in the discussion on graph coarsening in Chapter 5. We also implemented the an exact sparsest cut algorithm using enumeration, a graph traversal using the smallest last algorithm [97] which did not have meaningful results, and a preliminary rerouting algorithm further discussed in the section on future work in Chapter 10

8.3. Development and Continuous Integration Environment

A continuous integration environment was created for this project to avoid regressions in the compilation process, and in some specific areas for automated testing. These tests were useful when changing the memory model used for the graph representation, due to the way that the BGL treats template parameters and the way that Qt handles object copies, especially in the UI. When migrating between raw pointers, references, copies, Qt smart pointers, and finally C++11 smart pointers, several crashes occurred during the process. The automated tests provided minimal examples of the crash so that they could be debugged easily. They also served to identify future crashes as the test suite proceeded before trying to debug them in the main UI.

The environment also included automated code-quality scanning with CPPCheck and clang-tidy to identify error-prone code to fix. The clang-tidy tool was further used to modernize the code against C++11, and later C++14, C++17, and eventually the C++20 draft. This step in the continuous integration environment was a step towards a modern configuration, similar to what is available in professional software development teams.

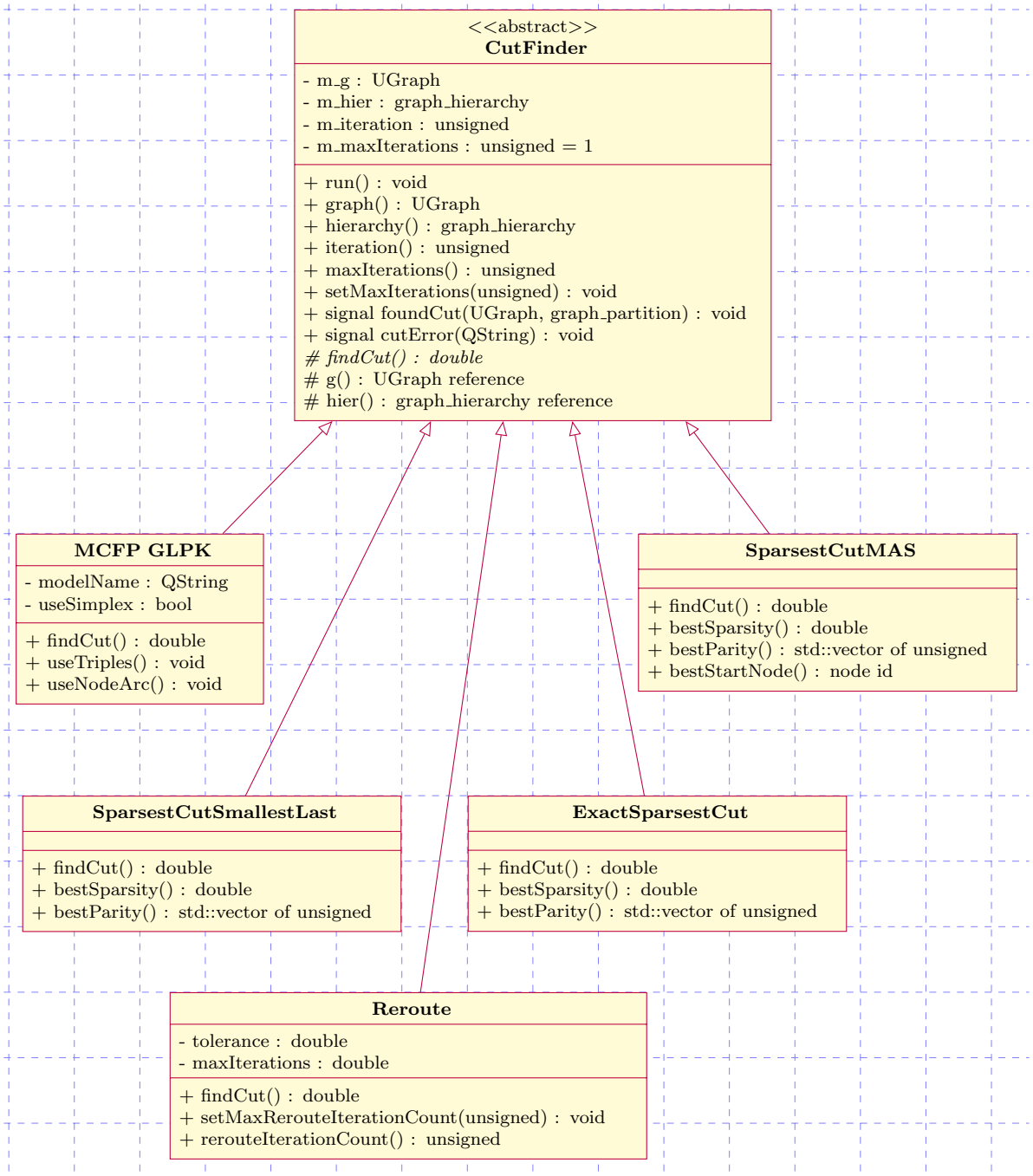


Figure 8.1. Class Diagram for Cut Finding Algorithms

Chapter 9

APPLICATIONS

In this paper so far, we have mentioned some applications, but focused on the some specific heuristics for the MCFP and SCP and the memory and runtime implications. In this chapter, we cover a few considerations for reduction to practice, and two potential application contexts.

9.1. Considerations in Reduction to Practice

When considering the potential applications of these heuristics, it is important to balance accuracy against runtime and memory utilization. While speed increased in processors for many years, recent history has focused less on the clock speed and more on accelerating applications via more efficient processor operations and on parallelization and vectorization (SIMD or MIMD) of those operations. Among other things, this led to the modification of the C++ Extensions for Parallelism from their original proposal in 2014 that focused on threading as a parallelization primitive. The version that was eventually adopted into C++17 included inputs from NVIDIA to consider add vectorization as an alternative parallelization primitive. The simplex algorithm is iterative, and threads may be used to explore alternative pivot vertices or input perturbations. However, vectorization approaches may provide significant speedup for the simplex algorithm matrix operations.

We also need to consider memory utilization. Even though modern server farms have shared memory over multiple cores and processor boards, it is hard to keep up with the growth of the MCFP. Any beginning course on big-O analysis covers how long operations will take as the algorithm scales. In memory growth, it is easy to hit this limit earlier, especially with the growth rates of the LP formulations of the MCFP. Additionally, if the computers executing the task use swap to augment the primary memory, it is easy to get

into a thrashing condition with most LP solvers as they update matrices.

It is also helpful to consider the impacts of bus throughput onto accelerator cards for these applications. Many libraries today can use GPU acceleration for matrix operations, requiring a data transfer from main memory into GPU memory and back, usually over a PCIe bus. Early attempts at this for investigation on flow rerouting for this project resulted in the transfers to and from GPU memory consuming more wall-clock time than the Floyd-Warshall APSP implementation on the GPU. This created a condition where the total time was similar to the pure CPU implementation. Since that time, the hardware industry has updated the PCIe standard, with other workloads encountering similar constraints. Nonetheless, it remains something to be considered when moving the volumes of data that the MCFP and even some of its rerouting heuristic approaches can create.

9.2. Social Network Analysis

SNA has become more and more prevalent over the years because of its ability to provide some idea of the problem space when the underlying data is either complex, ambiguous, or difficult to obtain, or some combination of these [27]. To reinforce this, consider that the efficient market hypothesis claims that all of the available underlying data is represented in the price of an asset. Then, sometimes the network structure might be the best available information.

Ronald Burt developed the concept of structural holes in [26], and showed them to be a more significant tool for analysis of social capital than network closure in [87]. If the concept of human capital is based on the unique abilities of an individual actor, then social capital represents the context in which to evaluate the human capital. Three are numerous examples of extremely talented individuals who stalled in their pursuits due to the context surrounding them. The authors of [87] reviewed two opposing views of network structure as a predictor of social capital, network closure, and structural holes.

The concept of network closure is based on the economic idea that within a closely knit group, information is redundant and shared quickly, thus creating a resistance to violating

the social norms of the group. When put into graph theory terms, these communities are represented as dense subgraphs, or portions of the graph where the individual nodes are highly interconnected. Here, social capital is increased by being a highly connected individual since any malfeasance would have a high probability of being detected. When considering competitive markets, a dense graph would considered efficient due to the high level of information flow.

In contrast, the subgraphs may be connected to each other through a series of possibly redundant bridges. A structural hole is present between the dense subgraphs when there are few (or no) connections between them. As an example, consider a complete graph on three nodes. If a single edge is removed, then a certain pair of nodes can only communicate through the third. Nodes that bridge between non-adjacent nodes in this was can function as information brokers, thus raising their social capital within a social network. Nodes that are information brokers have an advantage over others in that they have additional autonomy and bargaining negotiations. This also applies in competitive markets, where information asymmetries are considered to be a main reason for market inefficiency.

The concept of information brokering lends itself to a concept of information flow, intuitively. If link strength is represented as edge weight, then that weight could be used as a capacity on that edge to constrain the flow. Traditional attempts to identify broker nodes have centered around the measure of betweenness centrality introduced in [52]. To calculate this, an all-pairs shortest path algorithm is used to determine the number of shortest paths passing through each node. However, this does not account for the capacity of any edges that limit information exchange.

In [90], the authors introduce the concept of "flowthrough centrality", in which nodes that broker more information than others (as opposed to strictly receiving) would have a higher centrality than others. To calculate the flowthrough centrality, first the hierarchical maximum concurrent flow solution is found. Then the centrality measure is calculated as follows:

$$FT(v) = \frac{\sum_{\{u:(u,v)\in E\}} c_{u,v} - \sum_{w\in V} |f(w,v)|}{\sum_{\{u:(u,v)\in E\}} c_{u,v}}$$

where $c_{u,v}$ is the edge capacity on the edge between u and v, and $f(w,v)$ is the flow between v and every other node w. In other words, the sum of the adjacent capabilities less the flow originating or ending at v divided by the sum of the adjacent capacities.

While other centrality measures are not robust to edge removal, [90] argues that flowthrough centrality sees only a 16% penalty on average. This makes it a good measure for social network analysis (SNA), since it is rare that the "ground truth" graph is available, and a reasonable approximation is therefore used. It would be an interesting future study to apply the flowthrough centrality measure to a real-world problem similar to the job search problem used to evaluate network closure and structural holes in [86].

The main issue with the measurement of flowthrough centrality has to do with solving the hierarchical maximum concurrent flow problem (HMCFP). It is currently solvable through linear programming methods, requiring $O(mn^3)$ space and $O(n^{4.5})$ time per stage of the hierarchy. Even with warm starting each successive cut, the runtime for a full decomposition of the network takes considerable time for a large network. Worse, the high space constraint limits calculations to relatively small graphs.

Betweenness centrality requires only $O(n^2)$ space for the adjacencies and $O(n^3)$ time for the all-pairs shortest path algorithms. This can still be problematic for large graphs, but is more feasible than the requirements for flowthrough centrality because of the HMCFP size.

9.3. Compressed Sensing

The field of Compressed Sensing may be an area where the heuristics presented here provide some other value. Problems solvable using Compressed Sensing adhere to the "restricted isometry property", which allows $L0$ space (LP solution coefficient pseudo-norm) to be represented as $L1$ space (sum of LP solution coefficients) with minimal error. From each measurement, an underdetermined system of linear equations is created and the solution space found. Then, high probability, the sparsest solution (the one with the fewest nonzero

coefficients) is the correct answer [15]. This field provides some constraints not found in other areas like SNA. For instance, the problems can be scaled based on the desired resolution of the results, but they must be solved very quickly. It is not necessary to solve the hierarchical formulation, but the resulting resolution is based on both the problem size and the number of iterations executed to refine the solution. A key step in the algorithm is therefore finding a very sparse solution to the system. This must be done quickly, since finding these solutions is performed in a loop to create cutting planes that constrain the results in future observations.

The MCFP is a natural application here to estimate the sparsest cut, especially since most graphs are bottleneck graphs [57]. However, solving an LP in $O(n^{4.5})$ time in a tight loop for a guarantee of $O(\log n)$ can be more computationally intensive than desired. In ARV [11], an approach using expander flows and semi-definite programming was used to get $O(\sqrt{\log n})$ accuracy in asymptotically better time. However, the runtime of semi-definite program solvers is worse than most LP solvers in the general case. For this particular application, the approach in [11] executes in $O(n^2)$ time. Nonetheless, when the ARV approach to approximating the SCP solution came out, the approaches to the problem quickly became fast enough to encourage more researchers to work in the field. Since ARV supplanted MCFP as the SCP heuristic of choice for Compressed Sensing, the heuristics presented here may be able to further accelerate the field for certain use cases by giving a faster approach at the potential expense of accuracy.

9.4. Wildlife Management

Migratory corridors can be viewed as a spatial analog to social networks, allowing tools from SNA to be used. As such, centrality measures that show the influence some nodes have over others can be used to show the influence that sites might have over pathways [42]. As in other applications for the work presented in this document, the MCFP could enable a solution more tolerant to perturbations in the graph than the traditional approaches. The heuristics presented here would similarly allow faster processing of existing graphs or the

algorithm to process larger graphs than previously used. In this case, it could extend the work from a relatively small area in Australia to analyze larger migration patterns over more land area.

9.5. Medicine

Community structure based on EEGs may help to identify brain pathologies such as ADHD [2]. Initial approaches to finding community structure tend to use the Girvan-Newman betweenness centrality due to its common inclusion in math libraries. The "flowthrough centrality" from Mann may provide a more robust answer at the expense of additional computation time, especially as the size of the graph grows. This is an area where the heuristics presented here could help improve the runtime to calculate the flowthrough centrality to more quickly identify the community structure and key areas in the graphs used.

9.6. Transportation

The MCFP has previously been used to model traffic networks. Traffic patterns have been modeled as compressible flow networks once the models moved from queueing models, so the MCFP provides valuable insight into the flow as the roads become more congested. The work in [46] showed promise on smaller graphs but did not scale well. The triples formulation of the LP will likely help in finding solutions on some of these larger networks, and the heuristics presented in this document should help with speed or on even larger graphs than the LP.

Chapter 10

CONCLUSION

This paper has presented three heuristics to the MCFP and the beginnings of a quantum approach to the problem. These were all evaluated on a combination of example real and synthetic graphs. We also showed some implementation details for our approaches, to discuss some of the trades that should be considered as part of any reduction to practice.

10.1. Wrapping Up

We explored the use of the MAS as a heuristic to solving the MCFP and HMCFP and compared the performance in accuracy and computing resource utilization. We found that in most cases the heuristic finds the correct result with significantly lower runtimes and memory utilization.

This approach was extended to use the same heuristic to coarsen the graph before executing the MCFP LP on the coarsened graph. This also resulted in improved resource utilization. Because the resulting LP is smaller, it also enabled LP solutions on larger graphs than trying to run the LP without first coarsening.

Both of the above approaches were tested on a collection of graphs generated as RGGs on the unit square and unit sphere, random bipartite graphs, and random typing graphs. These graphs provided a large sample of synthetic graphs representing a variety of real-world graph types, while remaining tractable for the MCFP on commodity hardware.

We then introduced the D_3 bound to attempt to determine the MCFP throughput value without executing either the LP or an all-pairs shortest-paths algorithm. This approach was only tested on the collection of random bipartite graphs, though it was tested on graphs with a diameter greater than three with favorable results. The D_3 bound allows a heuristic bound that is as accurate as the shortest-path bound on these graphs when the minimum

degree exceeds eight without having to execure an APSP algorithm.

10.2. Open Questions and Future Areas of Study

So far, we have shown that if the shortest path bound is the maximum concurrent flow, then the graph is in a gridlocked condition. However, the opposite has not been proven. That is, any graph where where the shortest path bound is tight is a gridlocked graph, but there may exist gridlocked graphs for which the shortest path bound is not tight. Future work could include showing that graphs that reach gridlock also have a tight shortest path bound. This might be empirical from our generated graphs, or start there. The research may also be an analytical approach to prove a theorem that this is the case or show a counterexample.

One type of graph not explored here was the "half-random" geometric graphs. These are created using a grid of half the number of nodes, rounded to the nearest square number, then scattering the remainder randomly. The rest of the graph generation follows the same method as the RGG generation described above for the unit square. These graphs could serve to show what might happen in wireless sensor networks where some sensor distribution was preplanned for coverage, and other sensors distributed for redundancy.

Additionally, future could address the relation of RGG generation parameters to MCFP or SCP solutions. Building on the idea discussed in Chapter 2 where RGGs on the unit sphere reduce the boundary effects on the nodes in the graph, there could be investigations into what combinations of the number of nodes and radius keep the MCF cut "in the middle" of the graph versus causing a side or corner to get carved off. In terms of extending the heuristic analysis, future work would be to investigate what combinations of number of nodes and radius relate to the SCP solution matching the MCFP solution, and how that might impact the accuracy of the heuristic approaches discussed so far.

Another potential avenue of study is to use an existing LP solver with GPGPU features or to modify GLPK to support it. Once the most expensive step(s) of interior point methods is/are made faster, the Simplex algorithm may not perform as well in comparison for this type of data. Some attempts have been made at accelerating GLPK and LPSolve using both

NVIDIA CUDA and Khronos OpenCL, but none that have at this point be merged into the baseline. As covered in the discussion on libraries in Chapter 3, different libraries have been optimized for different LP solving algorithms, so any speedup that may come from GPU utilization may also depend on which approach is being accelerated. There is also the issue of maturity of any proposed patch to get the library maintainers to accept it into the baseline.

10.2.1. Flow Rerouting

Many network flow problems have been solved using flow rerouting algorithms, though sometimes better solutions were found with other approaches, for example the flow augmentation algorithms for min-cut like Ford-Fulkerson [50] and Edmonds-Karp [44] compared to the graph search approach from Stoer-Wagner [117]. Almost from the introduction of the MCFP in [112], flow rerouting approaches have been attempted, such as the ones in [19, 77, 112].

The approaches in Shahrokhi [112] and Biswas [19] papers were criticized in some early responses compared to other algorithms because of perceived inapplicability to the non-uniform capacity case. While the original papers assume a uniform case, they are derived from the non-uniform case with the claim of "without loss of generality". A new derivation without that assumption to create a rerouting algorithm would address this concern for the community. Also, the original formulations were only for the first cut in the hierarchy. Future work could include an extension to the hierarchical case.

The triples formulation in [41] intuitively has thoughts of flow rerouting. It would be interesting to see a rerouting approach based on this formulation to find a new heuristic that may be easily extensible to the hierarchical case. The exponential toll functions used in [19] could conceivably by extended to the triples concept for this new approach. This should create a heuristic that works on larger problems with better accuracy. If it only works for the first cut, it would still be a valuable contribution to the field. However, it would be more valuable if the heuristic applied to the hierarchical problem.

One final method of improving the LP performance in practice is to attempt to simplify the Simplex basis when problematic variables are added to the basis. There may be a method using the basis suppression approach suggested by Patty and Helgason in [104]. They previously used this approach on the MCFP, so it would be interesting to see how it performs both in modern LP solving libraries and in the hierarchical formulation of the MCFP. In the HMCFP, with or without warm-starting at each step, there may be different complicating variables between the steps that need to be removed from the basis.

10.2.2. LP Warmup

A few LP formulations of the Maximum Concurrent Flow Problem (MCFP) are discussed later. These formulations have large memory requirements and have slow execution times using either the Simplex or interior point methods. In order to reduce the run time, there has been significant research in the area of "warming up" an LP by providing a solution that is nearly optimal. [51, 68, 109, 125] However, finding a basic feasible solution to warm start a Simplex algorithm is often difficult since it amounts to finding an initial basis that is close to the optimal solution. Bixby's algorithm [20] was the best known approach for a while, with the approach in [71] providing a better basis in recent years. Most of the warm starting research has focused on interior point methods, which can have errors when provided an warm start solution too close to the boundary of the feasible polytope has been shown to cause the algorithm to take longer or even fail [116, 122]. It is therefore interesting to explore setting initial flows based on the MAS results. As an initial attempt, one should begin by setting the flows on critical edges to be uniformly distributed across the demand pairs of the cut, such that the edges remain critical. The experiment would then consist of an examination of the number of iterations required, and a qualitative look at whether the algorithm stalled during the optimization process.

So far, our implementation of the HMCFP restarts calculating the solution at each level in the hierarchy. The solution to the previous level cannot be used directly to warm start the new level, since the constraint matrix has changed. However, there may be a way to create

an initial basis for the new level from the previous solution, since both sets of constraints are known. Future work could include evaluating the benefits in runtime on the hierarchy, and whether that improvement is dependent on the LP library and algorithm being used.

Additionally, we have shown the ability to calculate heuristic solutions using MAS, both on its own, and as a coarsening function for an LP. Either of these approaches could also be used to warm start the initial step in the hierarchy. Similar questions to the hierarchical approach would be of interest in this future study.

Finally, there may be benefit in a hybrid approach. We have shown the ability to use the MAS heuristic in a hierarchical manner. If at each step in the hierarchy, we begin with the MAS heuristic and warm start the LP, then we should get the correct solution at each step. Future work would involve checking this hypothesis, and also evaluating the performance benefits of this approach.

BIBLIOGRAPHY

[1] Aharonov, D., Ambainis, A., Kempe, J., and Vazirani, U. Quantum walks on graphs. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2001), STOC '01, ACM, pp. 50–59. 109

[2] Ahmadlou, M., and Adeli, H. Functional community analysis of brain: A new approach for eeg-based investigation of the brain pathology. *NeuroImage 58*, 2 (2011), 401 – 408. 126

[3] Akoglu, L., and Faloutsos, C. Rtg: A recursive realistic graph generator using random typing. *DATA MINING AND KNOWLEDGE DISCOVERY 19*, 2 (2009), 194–209. 19, 21, 22, 54

[4] Albert, R., Jeong, H., and Barabási, A.-L. Diameter of the world-wide web. *nature 401*, 6749 (1999), 130–131. 21

[5] Albert, R., Jeong, H., and Barabási, A.-L. Error and attack tolerance of complex networks. *nature 406*, 6794 (2000), 378–382. 21

[6] Allalouf, M., and Shavitt, Y. Centralized and distributed algorithms for routing and weighted max-min fair bandwidth allocation. *IEEE/ACM Trans. Netw. 16*, 5 (Oct. 2008), 1015–1024. 92

[7] AMBAINIS, A. Quantum walks and their algorithmic applications. *International Journal of Quantum Information 01*, 04 (2003), 507–518. 109, 110, 111

[8] Ambainis, A., Kempe, J., and Rivosh, A. Coins make quantum walks faster. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2005), SODA '05, Society for Industrial and Applied Mathematics, pp. 1099–1108. 110

[9] Ambuhl, C., Mastrolilli, M., and Svensson, O. Inapproximability results for sparsest cut, optimal linear arrangement, and precedence constrained scheduling. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)* (Oct 2007), pp. 329–337. 5

[10] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammarling, S., Demmel, J., Bischof, C., and Sorensen, D. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 1990), Supercomputing ?90, IEEE Computer Society Press, p. 2?11. 34

[11] Arora, S., Rao, S., and Vazirani, U. Expander flows, geometric embeddings and graph partitioning. *J. ACM 56*, 2 (Apr. 2009). 7, 91, 125

[12] Aspnes, J., Chang, K., and Yampolskiy, A. Inoculation strategies for victims of viruses and the sum-of-squares partition problem. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2005), SODA '05, Society for Industrial and Applied Mathematics, pp. 43–52. 2

[13] Barabási, A.-L., and Albert, R. Emergence of scaling in random networks. *science 286*, 5439 (1999), 509–512. 21

[14] Barabasi, A.-L., and Posfai, M. *Network science*. Cambridge University Press, Cambridge, 2016. 54

[15] Baraniuk, R., Davenport, M. A., Duarte, M. F., and Hegde, C. *An Introduction to Compressive Sensing*. OpenStax CNX, 2014. 7, 125

[16] Bauguion, P.-O., Ben-Ameur, W., and Gourdin, E. Efficient algorithms for the maximum concurrent flow problem. *Networks 65*, 1 (2015), 56–67. 5

[17] Berkelaar, M., Eikland, K., and Notebaert, P. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, May 1 2004. Available at ¡http://lpsolve.sourceforge.net/5.5/¿. Last accessed Dec, 18 2009. 117

[18] Bienstock, D., and Raskina, O. Asymptotic analysis of the flow deviation method for the maximum concurrent flow problem. *Mathematical Programming 91* (2000), 2002. 5

[19] Biswas, J., and Matula, D. W. Two flow routing algorithms for the maximum concurrent flow problem. In *Proceedings of 1986 ACM Fall Joint Computer Conference* (Los Alamitos, CA, USA, 1986), ACM '86, IEEE Computer Society Press, pp. 629–636. 8, 11, 14, 42, 43, 60, 63, 92, 93, 112, 129

[20] Bixby, R. E. Implementing the simplex method: The initial basis. *ORSA Journal on Computing 4*, 3 (1992), 267–284. 33, 130

[21] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment 2008*, 10 (2008), P10008. 54

[22] Bollobas, B., and Klee, V. Diameters of random bipartite graphs. *Combinatorica 4*, 1 (1984), 7–19. 98, 104

[23] Bonsma, P. S. Linear time algorithms for finding sparsest cuts in various graph classes. *Electronic Notes in Discrete Mathematics 28* (2007), 265 – 272. 91

[24] Borgatti, S. P., and Everett, M. G. A graph-theoretic perspective on centrality. *Social Networks 28*, 4 (2006), 466 – 484. 7

[25] Broido, A. D., and Clauset, A. Scale-free networks are rare. *Nature communications 10*, 1 (2019), 1–10. 22

[26] Burt, R. S. Structure holes, 1992. 122

[27] Burt, R. S., et al. The social capital of structural holes. *The new economic sociology: Developments in an emerging field 148* (2002), 90. 122

[28] Cai, W., and Matula, D. W. Partitioning by maximum adjacency search of graphs. In Cox et al. [38], pp. 55–64. 37, 38, 56, 57

[29] Charikar, M., Chatziafratis, V., and Niazadeh, R. *Hierarchical Clustering better than Average-Linkage*. Society for Industrial and Applied Mathematics, 2019, pp. 2291–2304. 31

[30] Chatziafratis, V., Niazadeh, R., and Charikar, M. Hierarchical clustering with structural constraints. *CoRR abs/1805.09476* (2018). 31

[31] Chen, S.-J., and Cheng, C.-K. Tutorial on vlsi partitioning. *VLSI Design 11*, 3 (2000), 175–218. 3, 6

[32] Chen, Z., and Matula, D. W. Bipartite grid partitioning of a random geometric graph. In *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS)* (June 2017), pp. 163–169. 8, 19, 20

[33] Chiou, S.-W. A combinatorial approximation algorithm for concurrent flow problem and its application. *Comput. Oper. Res. 32*, 4 (Apr. 2005), 1007–1035. 92

[34] Chuzhoy, J., and Khanna, S. Polynomial flow-cut gaps and hardness of directed cut problems. *J. ACM 56*, 2 (Apr. 2009). 91

[35] Clauset, A., Newman, M. E. J., , and Moore, C. Finding community structure in very large networks. *Physical Review E* (2004), 1– 6. 54

[36] Clauset, A., Tucker, E., and Sainz, M. The colorado index of complex networks. *Retrieved July 20* (2016), 2018. 22

[37] Cohen, M. B., Lee, Y. T., and Song, Z. Solving linear programs in the current matrix multiplication time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing* (New York, NY, USA, 2019), STOC 2019, Association for Computing Machinery, p. 938?942. 4

[38] Cox, I. J., Hansen, P., and Julesz, B., Eds. *Partitioning Data Sets, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, April 19-21, 1993* (1995), vol. 19 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, DIMACS/AMS. 134

[39] Derici, I. M., and Panu, M. T. Determining the maximum clique size in large random geometric graphs. In *2011 International Conference on High Performance Computing Simulation* (July 2011), pp. 143–154. 8, 19

[40] Dong, T., Haidar, A., Tomov, S., and Dongarra, J. A fast batched cholesky factorization on a gpu. In *2014 43rd International Conference on Parallel Processing* (Sep. 2014), pp. 432–440. 35

[41] Dong, Y., Olinick, E. V., Jason Kratz, T., and Matula, D. W. A compact linear programming formulation of the maximum concurrent flow problem. *Networks 65*, 1 (2015), 68–87. 4, 5, 8, 27, 28, 31, 34, 92, 99, 129

[42] Dunn, A. G., and Majer, J. D. Measuring connectivity patterns in a macro-corridor on the south coast of western australia. *Ecological Management & Restoration 10*, 1 (2009), 51–57. 125

[43] Dürr, C., Heiligman, M., HOyer, P., and Mhalla, M. Quantum query complexity of some graph problems. *SIAM J. Comput. 35*, 6 (June 2006), 1310–1328. 108

[44] Edmonds, J., and Karp, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM 19*, 2 (Apr. 1972), 248–264. 1, 129

[45] Erdös, P., and Renyi, A. On random graphs i. *Publ. Math. Debrecen 6* (1959), 290–297. 19, 96

[46] Etemadnia, H., Abdelghany, K. F., and Hariri, S. Toward an autonomic architecture for real-time traffic network management. *Journal of Intelligent Transportation Systems 16* (2012), 45 – 59. 126

[47] Flake, G. W., Tarjan, R. E., and Tsioutsiouliklis, K. Graph clustering and minimum cut trees. *Internet Math. 1*, 4 (2003), 385–408. 54

[48] Floyd, R. W. Algorithm 97: Shortest path. *Commun. ACM 5*, 6 (June 1962), 345–. 100

[49] Ford, D. R., and Fulkerson, D. R. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 2010. 2

[50] Ford, Jr., L. R., and Fulkerson, D. R. Maximal flow through a network. *j-CAN-J-MATH 8*, ?? (???? 1956), 399–404. 2, 129

[51] Forsgren, A. On warm starts for interior methods. In *System Modeling and Optimization* (Boston, MA, 2006), F. Ceragioli, A. Dontchev, H. Futura, K. Marti, and L. Pandolfi, Eds., Springer US, pp. 51–66. 130

[52] Freeman, L. C. A set of measures of centrality based on betweenness. *Sociometry 40*, 1 (1977), 35–41. 123

[53] Gahrouei, A. R., and Ghatee, M. Effective implementation of gpu-based revised simplex algorithm applying new memory management and cycle avoidance strategies. *arXiv preprint arXiv:1803.04378* (2018). 35

[54] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*, first edition ed. W. H. Freeman, 1979. 2, 54

[55] Girvan, M., and Newman, M. E. J. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences 99*, 12 (2002), 7821–7826. 3, 54

[56] Goldberg, A. V., and Tarjan, R. E. A new approach to the maximum-flow problem. *Journal of the ACM (JACM) 35*, 4 (1988), 921–940. 38

[57] Gourdin, E. A mixed integer model for the sparsest cut problem. *Electronic Notes in Discrete Mathematics 36* (2010), 111 – 118. ISCO 2010 - International Symposium on Combinatorial Optimization. 2, 4, 33, 109, 125

[58] Grover, L. K. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1996), STOC '96, ACM, pp. 212–219. 108

[59] Guennebaud, G., Jacob, B., et al. Eigen v3. http://eigen.tuxfamily.org, 2010. 34, 117

[60] Gurung, A., and Ray, R. Simultaneous solving of batched linear programs on a gpu. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2019), ICPE ?19, Association for Computing Machinery, p. 59?66. 35

[61] Guttmann-Beck, N., and Hassin, R. Approximation algorithms for minimum k-cut. *Algorithmica 27*, 2 (2000), 198–207. 2

[62] Hajiaghayi, M., Johnson, T., Khani, M. R., and Saha, B. Hierarchical graph partitioning. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2014), SPAA '14, ACM, pp. 51–60.

[63] Hajiaghayi, M. T., and Räcke, H. An $\mathcal{O}(\sqrt{\log n})$-approximation algorithm for directed sparsest cut. *Information Processing Letters 97*, 4 (2006), 156 – 160. 106

[64] Hoenen, A., and Schenk, N. Knowing the author by the company his words keep. In *LREC* (2018). 22

[65] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering 9*, 3 (2007), 90–95. 118

[66] Johansson, J., Nation, P., and Nori, F. Qutip: An open-source python framework for the dynamics of open quantum systems. *Computer Physics Communications 183*, 8 (2012), 1760 – 1772. 112, 118

[67] Johansson, J., Nation, P., and Nori, F. Qutip 2: A python framework for the dynamics of open quantum systems. *Computer Physics Communications 184*, 4 (2013), 1234 – 1240. 112, 118

[68] John, E., and Yildirim, E. A. Implementation of warm-start strategies in interior-point methods for linear programming in fixed dimension. *Computational Optimization and Applications 41*, 2 (Nov 2008), 151–183. 130

[69] Jung, J. H. Cholesky decomposition and linear programming on a gpu. Master's thesis, University of Maryland, 2006. 35

[70] Jung, J. H., and O'Leary, D. P. Implementing an interior point method for linear programs on a cpu-gpu system. *ETNA. Electronic Transactions on Numerical Analysis [electronic only] 28* (2007), 174–189. 35

[71] Junior, H. V., and Lins, M. P. E. An improved initial basis for the simplex algorithm. *Computers & Operations Research 32*, 8 (2005), 1983–1993. 33, 130

[72] Karmarkar, N. A new polynomial-time algorithm for linear programming. *Combinatorica 4*, 4 (Dec 1984), 373–395. 4

[73] Kempe, J. Quantum random walks hit exponentially faster. *CoRR quant-ph/0205083* (2002). 112

[74] Kempe, J. Quantum random walks: An introductory overview. *Contemporary Physics 44*, 4 (2003), 307–327. 109

[75] Khot, S. On the power of unique 2-prover 1-round games. In *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2002), STOC '02, ACM, pp. 767–775. 5

[76] Khot, S. A., and Vishnoi, N. K. The unique games conjecture, integrality gap for cut problems and embeddability of negative type metrics into ℓ 1. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science* (Washington, DC, USA, 2005), FOCS '05, IEEE Computer Society, pp. 53–62. 5

[77] Klein, P., Plotkin, S., Stein, C., and Tardos, E. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM J. Comput. 23*, 3 (June 1994), 466–487. 129

[78] Lalami, M. E., El-Baz, D., and Boyer, V. Multi gpu implementation of the simplex algorithm. In *2011 IEEE International Conference on High Performance Computing and Communications* (Sep. 2011), pp. 179–186. 35

[79] Le Boudec, J.-Y. Rate adaptation, congestion control and fairness: A tutorial. *Web page, November* (2005). 4

[80] Lee, Y. T., Song, Z., and Zhang, Q. Solving empirical risk minimization in the current matrix multiplication time. In *COLT* (2019). 4

[81] Leighton, T., and Rao, S. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM 46*, 6 (Nov 1999), 787–832. 3, 91, 92

[82] Leskovec, J., Kleinberg, J., and Faloutsos, C. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data 1*, 1 (Mar. 2007), 2?es. 18

[83] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014. 18

[84] Leskovec, J., Rajaraman, A., and Ullman, J. D. *Mining of Massive Datasets*, 2nd ed. Cambridge University Press, USA, 2014. 6

[85] Li, T., Li, H., Liu, X., Zhang, S., Wang, K., and Yang, Y. Gpu acceleration of interior point methods in large scale svm training. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (July 2013), pp. 863–870. 35

[86] Lin, N. A network theory of social capital. *The handbook of social capital 50*, 1 (2008), 69. 124

[87] Lin, N., Cook, K. S., and Burt, R. S., Eds. *Social Capital: Theory and Research*. Aldine de Gruyter, New York, 2001. 122

[88] Machiavelli, N. *The Prince, etc.*, vol. 1232 of *Project Gutenberg*. Project Gutenberg, P.O. Box 2782, Champaign, IL 61825-2782, USA, 1998. 17

[89] Mahjoub, D., and Matula, D. W. Employing (1-ϵ) dominating set partitions as backbones in wireless sensor networks. In *2010 Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments (ALENEX)* (2010), SIAM, pp. 98–111. 8, 19

[90] Mann, C. F. *Extensions of Maximum Concurrent Flow to Identify Hierarchical Community Structure and Hubs in Networks*. PhD thesis, Southern Methodist University, Dallas, TX, USA, 2008. AAI3303902. 6, 7, 17, 29, 30, 31, 60, 123, 124

[91] Mann, C. F., Matula, D. W., and Olinick, E. V. The use of sparsest cuts to reveal the hierarchical community structure of social networks. *Social Networks 30*, 3 (2008), 223 – 234. 3, 6, 7, 42, 54, 60, 64

[92] Matula, D. W. Cluster validity by concurrent chaining. In *Numerical Taxonomy*, J. Felsenstein, Ed., vol. 1 of *NATO ASI Series*. Springer Berlin Heidelberg, 1983, pp. 156–166. 3, 92

[93] Matula, D. W. Concurrent flow and concurrent connectivity on graphs. In *Graph Theory with Applications to Algorithms and Computer Science*, Y. Alavi, G. Chartrand, D. R. Lick, C. E. Wall, and L. Lesniak, Eds. John Wiley & Sons, Inc., New York, NY, USA, 1985, pp. 543–559. 2, 3, 19, 93, 94

[94] Matula, D. W. Divisive vs. agglomerative average linkage in hierarchical clustering. In *Classification as a tool of research: proceedings of the 9th Annual Meeting of the Classification Society (F.R.G.) University of Karlsruhe, F.R.G., 26-28 June, 1985* (1986), C. S. Meeting, W. Gaul, and M. Schader, Eds., North-Holland, pp. 318–334. 1, 3, 12, 26, 54, 92

[95] Matula, D. W. A linear time 2-plus-ϵ approximation algorithm for edge connectivity. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms* (Philadelphia, PA, USA, 1993), SODA '93, Society for Industrial and Applied Mathematics, pp. 500–504. 37, 56, 115

[96] Matula, D. W., and Dolev, D. Path-regular graphs. Tech. rep., Stanford University, Department of Computer Science, June 1980. 94

[97] Matula, D. W., Marble, G., and Isaacson, J. D. Graph coloring algorithms. In *Graph theory and computing*. Elsevier, 1972, pp. 109–122. 119

[98] Matula, D. W., and Shahrokhi, F. Sparsest cuts and bottlenecks in graphs. *Discrete Applied Mathematics 27*, 1-2 (1990), 113 – 123. 1, 2, 3, 5, 41, 56, 93, 107

[99] McCarthy, A. D. *Gridlock in networks: the Leximin method for hierarchical community detection*. PhD thesis, Southern Methodist University, 2017. 95

[100] Nace, D., and Pioro, M. A tutorial on max-min fairness and its applications to routing, load-balancing and network design. In *4th IEEE International Conference on Computer Sciences Research, Innovation and Vision for the Future (RIVF 2006)* (04 2006). 4

[101] Newman, M. E. J., and Girvan, M. Finding and evaluating community structure in networks. *Phys. Rev. E 69* (Feb 2004), 026113. 23

[102] Okamura, H., and Seymour, P. D. Multicommodity flows in planar graphs. *Journal of Combinatorial Theory, Series B 31*, 1 (1981), 75–81. 4

[103] Oliphant, T. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed ¡today¿]. 118

[104] Patty, B. W., and Helgason, R. V. The basis suppression method. *Annals of Operations Research 20*, 1 (Dec 1989), 233–248. 130

[105] Percival, C. Stronger key derivation via sequential memory-hard functions. *Self-published* (2009), 1–16. 5

[106] Ploskas, N., and Samaras, N. Gpu accelerated pivoting rules for the simplex algorithm. *Journal of Systems and Software 96* (2014), 1–9. 35

[107] Ploskas, N., and Samaras, N. Efficient gpu-based implementations of simplex type algorithms. *Applied Mathematics and Computation 250* (2015), 552–570. 35

[108] Porter, M. A., Onnela, J.-P., and Mucha, P. J. Communities in Networks. *Notices of the AMS* (Sept. 2009). 23

[109] Ralphs, T. K. Duality and warm starting in integer programming. 130

[110] Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., and Kepner, J. Survey and benchmarking of machine learning accelerators. *2019 IEEE High Performance Extreme Computing Conference (HPEC)* (Sep 2019). 34

[111] Scott, J. P. *Social Network Analysis: A Handbook*. SAGE Publications, Jan. 2000. 7

[112] Shahrokhi, F., and Matula, D. W. The maximum concurrent flow problem. *J. ACM 37*, 2 (Apr. 1990), 318–334. 1, 2, 3, 4, 25, 26, 27, 42, 91, 92, 93, 94, 108, 129

[113] Siek, J., Lee, L.-Q., and Lumsdaine, A. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 38, 100, 117

[114] Smith, J., and Mosca, M. Algorithms for quantum computers. In *Handbook of Natural Computing*. Springer, 2012, pp. 1451–1492. 108, 109

[115] Spampinato, D. G., and Elstery, A. C. Linear optimization on modern gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (2009), IEEE, pp. 1–8. 35

[116] Sporre, G., and Forsgren, A. Relations between divergence of multipliers and convergence to infeasible points in primal-dual interior methods for nonconvex nonlinear programming by göran sporre and anders forsgren optimization and systems theory. 130

[117] Stoer, M., and Wagner, F. A simple min-cut algorithm. *J. ACM 44*, 4 (July 1997), 585–591. 8, 37, 38, 56, 115, 116, 129

[118] Thompson, B. H., and Matula, D. W. A flow rerouting algorithm for the maximum concurrent flow problem with variable capacities and demands, and its application to cluster analysis. Tech. rep., Southern Methodist University, Computer Science Department, March 1986. 11, 15

[119] Vilas, F., Olinick, E., and Matula, D. W. A o(nm) heuristic to the hierarchical maximum concurrent flow problem. In *AIAA Information Systems-AIAA Infotech @ Aerospace* (2017). 37, 77

[120] Vilas, F. E., Olinick, E., and Matula, D. Graph coarsening for runtime improvements in the maximum concurrent flow problem. In *2018 AIAA Information Systems-AIAA Infotech @ Aerospace* (2018), AIAA SciTech Forum, American Institute of Aeronautics and Astronautics. 53

[121] Vilas, F. E., Olinick, E. V., and Matula, D. W. Bounds on maximum concurrent flow in random bipartite graphs. *Optimization Letters* (2020). 90

[122] Wächter, A., and Biegler, L. T. Failure of global convergence for a class of interior point methods for nonlinear programming. *Mathematical Programming 88*, 3 (Sep 2000), 565–574. 130

[123] Warshall, S. A theorem on boolean matrices. *J. ACM 9*, 1 (Jan. 1962), 11–12. 100

[124] Yang, D., Sun, J., Lee, J., Liang, G., Jenkins, D., Peterson, G., and Li, H. Performance comparison of cholesky decomposition on gpus and fpgas. In *Symposium on Application Accelerators in High Performance Computing* (07 2010). 35

[125] Yildirim, E. A., and Wright, S. J. Warm-start strategies in interior-point methods for linear programming. *SIAM Journal on Optimization 12*, 3 (2002), 782–810. 130

[126] Yin, H., Benson, A. R., Leskovec, J., and Gleich, D. F. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2017), KDD ?17, Association for Computing Machinery, p. 555?564. 18

[127] Zheng, A., Dunagan, J., and Kapoor, A. Active graph reachability reduction for network security and software engineering. In *IJCAI'11 Proceedings of the Twenty-Second international joint conference on Artificial Intelligence* (July 2011), AAAI Press, pp. 1750–1756. 2, 6

[128] Zipf, G. K. *Selective Studies and the Principle of Relative Frequency in Language.* Harvard University Press, 1932. 21