RICE UNIVERSITY

Novel Techniques for the Zero-Forcing and p-Median Graph Location Problems

by

Caleb C. Fast

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Illya V. Hicks, Chair
Professor of Computational and Applied Mathematics

Robert E. Bixby
Noah Harding Professor Emeritus of Computational and Applied Mathematics

---

Keith D. Cooper
L. John and Ann H. Doerr Professor of Computational Engineering

---

Richard A. Tapia
University Professor, Maxfield-Oshman Professor in Engineering

Houston, Texas

May, 2017

ABSTRACT

Novel Techniques for the Zero-Forcing and p-Median Graph Location Problems

by

Caleb C. Fast

This thesis presents new methods for solving two graph location problems, the p-Median problem and the zero-forcing problem. For the p-median problem, I present a branch decomposition based method that finds the best p-median solution that is limited to some input support graph. The algorithm can either be used to find an integral solution from a fractional linear programming solution, or it can be used to improve on the solutions given by a pool of heuristics. In either use, the algorithm compares favorably in running time or solution quality to state-of-the-art heuristics.

For the zero-forcing problem, this thesis gives both theoretical and computational results. In the theoretical section, I show that the branchwidth of a graph is a lower bound on its zero-forcing number, and I introduce new bounds on the zero-forcing iteration index for cubic graphs. This thesis also introduces a special type of graph structure, a *zero-forcing fort*, that provides a powerful tool for the analysis and modeling of zero-forcing problems.

In the computational section, I introduce multiple integer programming models for finding minimum zero-forcing sets, and integer programming and combinatorial branch and bound methods for finding minimum connected zero-forcing sets. While the integer programming methods do not perform better than the best combinatorial method for the basic zero-forcing problem, they are easily modified to enforce connectivity, and they are the best methods for the connected zero-forcing problem.

Acknowledgements

For any project I may undertake, the outcome of that project is known and ordained beforehand by my Creator. Consequently, any listing of acknowledgements would be incomplete without thanking God for His providence, in which I completed this work. In addition, mathematics and engineering are attempts to understand and exploit the order instituted at creation. Thus, I must thank God for creating a world in which mathematics could be studied and placing me in a position to study it.

I would like to thank my advisor, Illya V. Hicks, for his support throughout the long process of writing this thesis. His encouragement and belief in my ability helped me through the times when I did not see a way forward in my research. Additionally, I would like to thank him for the life experience, wisdom, and leadership lessons that he has passed on to me during my time as his student.

I would like to thank my beloved wife, Bethany, for her support through this time. From beginning to end of my graduate studies, she has lifted me up when I was despondent, provided me motivation to keep studying, and done everything she could to assist me in this effort.

I would also like to thank my father, under whose teaching I first learned of the beauty and challenge of mathematics; my commitee (Richard Tapia, Robert Bixby, and Keith Cooper), for guiding my research into productive and marketable directions; Randy Davila, for introducing me to the zero-forcing problem; and Boris Brimkov, for helpful discussion and collaboration on the zero-forcing problem. The research on these pages was funded through an ExxonMobil Graduate Fellowship, a Lodieska Stockbridge Vaughn Fellowship, and NSF grants CMMI-1300477 and CMMI-1404864.

Contents

Illustrations

Tables

Chapter 1

Introduction

This thesis explores new ways that combinatorial information can be combined with integer programming information to solve the p-median and zero-forcing problems. In particular, for the p-median problem, this thesis introduces a new algorithm based on branch decompositions that is able to provide close to optimal integer solutions from a linear programming relaxation and is also able to improve the quality of a pool of sub-optimal heuristic solutions. For the zero-forcing problem, this thesis provides bounds on the zero-forcing number and zero-forcing iteration index that are based on the branchwidth of the graph and combinatorial structures within the graph. This thesis also introduces new integer programming formulations for the zero-forcing and connected zero-forcing problems.

This thesis is organized as follows. Chapter 1 introduces the problems to be studied as well as some preliminary notations and techniques that will be used throughout. Chapter 2 reviews the existing literature related to the p-median problem, and Chapter 3 introduces a new branch decomposition based heuristic for the p-median problem. Chapter 4 reviews the existing literature for the zero-forcing problem. Chapter 5 presents new bounds on the zero-forcing number and iteration index of a graph, and Chapter 6 introduces an integer programming strategy for finding minimum cardinality zero-forcing sets. Chapter 7 summarizes the results and offers some interesting directions for future research.

The following sections of this introduction present the problems addressed in

this thesis. Section 1.1 explains the terminology that will be used throughout this thesis. Section 1.2 introduces the p-median problem, and Section 1.3 introduces the zero-forcing problem. Section 1.4 introduces branch decompositions, a major tool that I will use throughout this thesis.

1.1 Preliminary Graph Theory Definitions

I will use the following terminology in this thesis, a graph G consists of a set V of vertices and a set $E \subset V \times V$ of edges. A graph can be simple, in which case there are no edges that begin and end on the same vertex, i.e. $\forall \{v, w\} \in E$, $v \neq w$, and there are not multiple edges between the same pair of vertices, i.e. $\forall e_1, e_2 \in E$, $e_1 \neq e_2$. A graph can also be either directed or undirected. The edges of a directed graph are ordered pairs, whereas the edges of an undirected graph are simply sets. Any simple directed graph has an underlying simple undirected graph that is obtained by simply removing the direction from the edges (the order from the ordered pairs), and removing any duplicate edges formed in the process. If $H = (V_H, E_H)$ and $G = (V_G, E_G)$ are graphs, then H is a *subgraph* of G if $V_H \subset V_G$ and $E_H \subset E_G$. If E_H is maximal, that is if $E_H = \{\{a, b\} | a, b \in V_H, \{a, b\} \in E_G\}$, then H is called a *vertex induced* subgraph of G, or simply an *induced* subgraph of G. If H is a subgraph (or induced subgraph) of G, then G is said to *contain* H as a subgraph (or induced subgraph). For a set of edges $E_H \subset E_G$, there is a corresponding set of vertices V_H that contains all the vertices that are in an edge of E_H. The graph $H = (V_H, E_H)$ is the subgraph induced by the edge set E_H.

For a given vertex v of an undirected graph, if there exists an edge $\{v, w\} \in E$, then we call w a *neighbor* of v. The *open neighborhood* of a vertex is the set of all neighbors of that vertex. The *closed neighborhood* also includes the vertex itself.

The *degree* of a vertex is the number of neighbors of that vertex, i.e. the cardinality of the open neighborhood of the vertex.

An undirected graph in which every vertex has the same degree is called a *regular* graph or a *k-regular* graph, where k is the degree of a vertex in the graph. Some regular graphs have special names. A 2-regular graph is a *cycle*, and a 3-regular graph is a *cubic* graph. A *tree* is a graph that does not contain an edge-induced subgraph that is a cycle. A *leaf* of a tree is a vertex in the tree that has a degree of 1. All other vertices of the tree are called *interior* vertices.

1.2 The p-Median Problem*

The p-Median Problem (PMP) falls into the general category of facility location problems, and it is one of the four most important problems in facility location [109]. Given a set of locations that serve as both customer locations and potential facility locations and given a set of costs associated with serving each customer from each potential facility location, the PMP asks for the set of p facility locations that minimizes the cost of serving the locations that are not chosen. The PMP can also be stated in graph theoretical terms. Given a simple, undirected graph with weighted edges, the PMP asks for the set of vertices, M, with $|M| = p$ and a set of edges, A, connecting each vertex not in M to a vertex in M such that the total weight of the edges chosen to be in A is minimized. In industrial problems, it may be necessary to add additional constraints to the problem. For example, each facility may have a capacity so that there is a maximum number of edges in A that can be incident to the same vertex, or there may also be a cost associated with

*This section is adapted from [68].

choosing each particular vertex to be in M. However, in this thesis, I limit my consideration to the basic problem without any additional constraints.

For the p-median problem, this thesis introduces a new algorithm based on branch decompositions that is able to provide close to optimal integer solutions from a linear programming relaxation and is also able to improve the quality of a pool of sub-optimal heuristic solutions. The edges in a solution to the linear programming relaxation or the pool of heuristic solutions can be combined to give a graph. My algorithm decomposes this graph using a branch decomposition. Then, I use dynamic programming to build up a complete solution from the solutions on the leaves of the decomposition. This algorithm works well and can beat state-of-the-art methods when the graphs given by the linear program or heuristic solutions allow a good branch decomposition.

1.3 The Zero-Forcing Problem [†]

Like the PMP, the Zero-Forcing Problem (ZFP), can be thought of as a facility location problem, and I will show in Chapter 6 that it is a minimum set cover problem. However, the most natural way to think of it is as a graph infection problem on a simple, undirected graph. As a graph infection problem, the ZFP obeys the following infection rule. An uninfected vertex becomes infected if it is the only uninfected neighbor of an infected vertex. A set of infected vertices that is capable of infecting the entire graph through repeated applications of the infection rule is called a *zero-forcing set*. The ZFP asks for the minimum cardinality zero-forcing set of a given graph.

[†]This section is adapted from [67].

The ZFP holds an interesting place in the population of graph infection models. Most infection models focus on conditions under which an uninfected vertex will become infected. On the other hand, the ZFP focuses on a condition under which an infected vertex can infect its neighbors. This difference can be seen by constrasting the ZFP with irreversible k-threshold processes such as the processes studied by Dreyer and Roberts [53]. In the k-threshold infection process, a vertex becomes infected if at least k of its neighbors are infected. There are at least two major differences between the ZFP and the k-threshold process. First, as previously mentioned, the k-threshold process focuses on the uninfected vertices and construct rules for becoming infected based on interaction with infected vertices, but the ZFP focuses on the infected vertices and a rule for how those vertices can spread the infection. This difference in focus leads to the second major difference. Where the k-threshold is constant throughout the graph, the threshold in ZFP changes from vertex to vertex depending on degree. Thus, in the ZFP, some vertices can be seen as more capable, or requiring less resources to spread the infection than others. Thus, the two models belong to two distinct categories of infection models. In fact, Amos, Caro, Davila, and Pepper [6] have introduced a k-threshold type generalization of zero-forcing where instead of requiring only one uninfected neighbor for infection, at most k uninfected neighbors are required.

The zero-forcing infection rule leads to two interesting graph invariants. The first is the size of a minimum zero-forcing set, called the *zero-forcing number*. The second is the minimum number of applications of the infection rule required to infect the entire graph from a minimum zero-forcing set, called the *zero-forcing iteration index*. Throughout this thesis, I will denote the zero-forcing number by $Z(G)$ and the iteration index by $I(G)$.

The iteration index quantity has taken on two different names in the literature. The term *iteration index* was introduced by Chilakamarri, Dean, Kang, and Yi [40], but Hogben et al. [83], in the same year, used the term *minimum propagation time*. To avoid confusion, I use the term *iteration index* to refer to the minimum number of iterations required to color the graph starting from a minimum zero-forcing set, and I use the term *propagation time* to refer to the minimum number of iterations required to color the graph from a specific zero-forcing set. Thus, the iteration index is an invariant of a given graph, but the propagation time is a property of a given zero-forcing set of the graph.

For the zero-forcing problem, this thesis provides bounds on the zero-forcing number and zero-forcing iteration index that are based on the branchwidth of the graph and combinatorial structures within the graph. In particular, I prove that the zero-forcing number of a graph is at least as large as the branchwidth of the graph, and I prove that any zero-forcing set must contain a vertex from every *fort*, which is a set of vertices in the graph that satisfies certain conditions. The bounds on the zero-forcing iteration index are based on the combinatorial structures within the graph. In particular, I prove that the zero-forcing iteration index of a cubic graph is at most $\frac{3}{4}$ of the number of vertices in the graph, and I prove that claws and leaves in any graph can be used to bound the iteration index. Finally, this thesis introduces three different integer programming formulations for the zero-forcing problem. I compare these formulations to my C++ implemenation of the Wavefront algorithm [35] which was previously only available in Sage [129]. Our results show that while integer programming methods do not perform as well as Wavefront for the zero-forcing problem, they are the best performing methods for solving the *connected* zero-forcing problem.

1.4 Branch Decompositions

Branch decompositions were introduced by Robertson and Seymour [113] in their study of the Graph Minor Theorem, and they have since proved to be an efficient means of solving NP-hard problems on certain graphs. For example, Cook and Seymour [41] used branch decompositions in a heuristic for the traveling salesman problem, and Hicks [81] used them to solve the graph minor containment problem. In this thesis, I use branch decompositions to solve the p-median problem, and to bound the size of a minimum zero-forcing set.

A *branch decomposition* of a graph is simply an assignment of the edges of the graph to the leaves of a tree such that every interior (i.e. non-leaf) vertex of the tree has degree 3. The subtrees of this tree provide a hierarchical division of the graph into the subgraphs induced by the edges assigned to the leaves of the subtrees. Such hierarchical decompositions are the foundation of dynamic programming algorithms. Figure 1.1 gives an example of a branch decomposition.

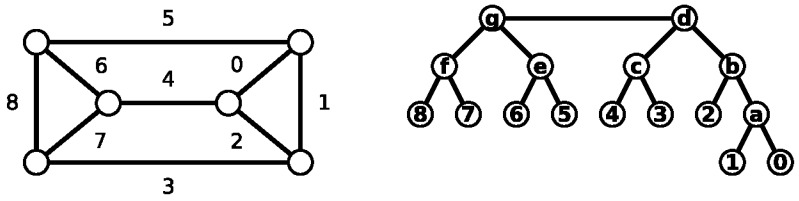


Figure 1.1 : An example of a branch decomposition. Each of the edges in the original graph (left graph) is assigned to one of the leaves of the the branch decomposition (right graph). Also, each interior vertex has degree exactly 3.

I now give a formal definition of a branch decomposition.

Definition 1.1 Consider a graph, $G = (V, E)$, and a tree, T, and denote the leaves of T by L. Suppose that each vertex of $T \backslash L$ has degree exactly 3, and suppose further there exists a bijection, $\tau : E \leftrightarrow L$. Then, the pair (T, τ) is a *branch decomposition* of G.

One of the uses of branch decompositions is to define the subproblems in a dynamic programming scheme. For this purpose, it is convenient to work with *rooted* branch decompositions. If a branch decomposition tree contains an edge, then the *rooted* version of that branch decomposition has one additional vertex with degree 2 that is designated as the root vertex of the tree. Any branch decomposition of a graph with more than one edge can be rooted by subdividing an edge of the tree and designating as the root the degree 2 vertex that is created by the subdivision. If a branch decomposition tree contains only a single vertex, then that vertex is designated as the root. If a branch decomposition tree is empty, then the rooted branch decomposition tree is also empty. Figure 1.2 gives an example of a rooted branch decomposition.

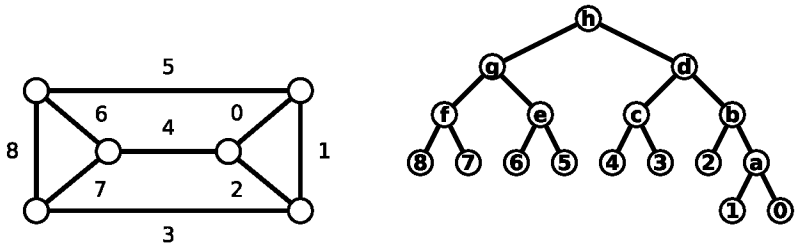


Figure 1.2 : An example of a rooted branch decomposition. The root (labeled h) has been inserted by subdividing the g-d edge. Each interior vertex except the root (labeled h) has degree exactly 3.

Dynamic programming works by breaking a problem into a hierarchy of

subproblems so that the solutions on the smaller subproblems can be reused and combined to form the solution to a larger subproblem. Qualitatively, the less interaction that exists between subproblems (i.e. the less vertices or edges of the graph that are shared by the subproblems) the less effort is required to merge the solutions of the subproblems to form a solution of a larger subproblem. Thus, the goal of decompositions, in terms of dynamic programming, is to minimize the interaction between different subproblems.

For branch decompositions, the interaction between different problems is measured by a property called *width*. Informally, the width of a branch decomposition is the maximum number of vertices that a subtree of the branch decomposition shares with the rest of the branch decomposition tree. Figure 1.3 gives an example of how subtrees share vertices. More formally, for a given subtree of the decomposition, the set of nodes of G that have incident edges from both the subgraph defined by the subtree and the subgraph defined by the rest of the decomposition is called the *middle set* of the subtree (These middle sets are the black vertices in Figure 1.3). In this thesis, I will identify the middle set of a subtree by the edge that joins the two branches of the decomposition tree. The *width* of a given branch decomposition is the cardinality of its largest middle set (i.e. the maximum cardinality of the middle set of an edge of the branch decomposition tree). The *branchwidth* of a graph is the minimum width over all branch decompositions of that graph. Thus, branchwidth is a graph invariant. In some cases, the branchwidth provides a bound on other graph invariants. In particular, I show in this thesis that branchwidth provides a lower bound on the zero-forcing number.

Obviously, there are multiple possible branch decompositions for non-trivial graphs, but for the purpose of developing a dynamic programming algorithm, the

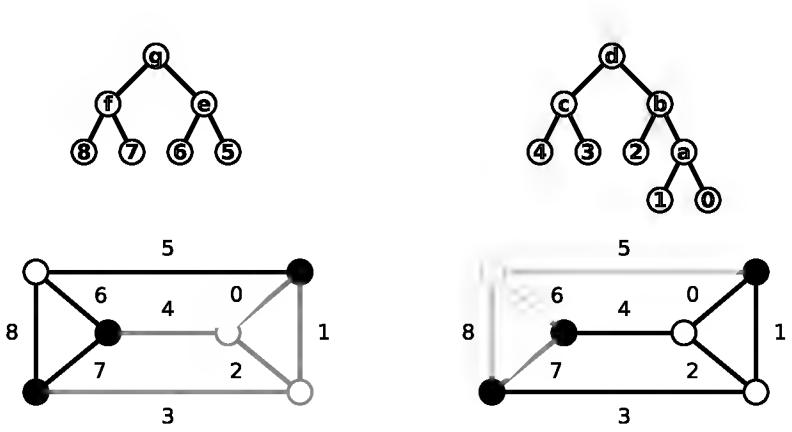


Figure 1.3 : An example of vertices shared by decomposition subtrees. The bottom graphs show the subgraphs induced (dark edges) by the subtrees above them. The black vertices are shared by both subgraphs. The full graph and branch decomposition are in Figure 1.1.

best branch decomposition to use is the one with smallest width. Consequently, the focus of algorithms for finding branch decompositions is to minimize width. Ideally, the width of a branch decomposition will be equal to the branchwidth of the graph. However, while finding such a minimum width branch decomposition can be done in polynomial time for planar graphs, as shown by Seymour and Thomas [125] and implemented by Hicks [82], finding such decompositions for general graphs is NP-hard [125]. However, this fact does not preclude the development of effective algorithms based on branch decompositions because the width of the branch decomposition only affects the efficiency of the algorithm based on it and not the quality of the solution that it finds. In other words, even if an algorithm is run with two different branch decompositions, the solution produced by the algorithm will be the same. The particular decomposition that is used will only affect the running time of the algorithm. Thus, a branch decomposition based algorithm can use a

branch decomposition obtained by heuristic methods. Such heuristics allow a "good enough" branch decomposition to be found quickly and then used to solve the original problem. If the heuristic branch decomposition is close to optimal, then the time required to prove optimality for the branch decomposition will likely surpass any time saved by using a potentially better branch decomposition. Therefore, throughout this thesis, I use a heuristic to find branch decompositions. Hicks [80] developed the branch decomposition heuristic that I use in this thesis.

Chapter 2

p-Median Literature*

The p-Median Problem (PMP) is useful in a wide variety of applications, and, as is the case with other important facility location problems, it is NP-hard [88]. The PMP was introduced by Hakimi [78] in 1965 to distribute switching centers in a communication network. The most common integer programming formulation for the problem, and the formulation I use in this thesis, was introduced by ReVelle and Swain [112] in 1970. These authors were studying facility location problems with the goal of minimizing the average distance between a fixed number of chosen supply points (the medians) and the given demand points. In general, the PMP is useful for facility location problems when temporal or political constraints preclude the ability to increase the number of facilities. For example, given a limited amount of medical supply units, Kunkel, Van Itallie, and Wu [92] used the PMP to distribute these units to villages in Malawi. In this case, the number, p, of medians that must be chosen is given by the number of medical supply units, and the available locations for placement are given by the location of villages in Malawi.

In addition to its basic form, the p-median problem has also been modified for similar problems. For example, while many facilities, such as medical supplies, are desirable and should be located as close as possible to customers, other facilities, such as sewage treatment facilites or chemical plants, are not desirable and should be located as far as practically possible from customers. Such undesirable facilities

*This chapter is expanded from [68].

are labeled "obnoxious," and the problem of locating them is called the *obnoxious p-median problem*, see for example Welch and Salhi [134]. Tamir [127] showed that, like the PMP, the obnoxious PMP is also NP-hard. Another possible circumstance is that some number of facilities are already existing, and some specified number of facilities need to be added. This circumstance is called the *conditional p-median problem* or the *(p,q)-median problem* where q is the number of existing facilities, see for example Minieka [99] or Drezner [55]. A related possibility is that additional facilities will become available at certain points over a time interval. The problem of placing the facilities to minimize distance over the whole time interval is the *progressive p-median problem*, see for example Drezner [54]. However, in this thesis, I focus on the basic PMP.

The PMP is also useful for data clustering. In the data clustering context, the PMP is sometimes called the k-medoid problem. The data clustering application is easy to understand since it simply requires reinterpreting the medians as the median of a cluster instead of as a facility. Mulvey and Crowder [102] used the p-median model for data clustering in this way. Ng and Han [104] showed that the p-median model was useful for detecting patterns and mining data as well as clustering. Hansen, Brimberg, Urošević, and Mladenović [79] noted that the p-median model is very general since the the distance measure that is used can be changed based on the practicioners preference. For example, by replacing the pairwise distance between points by the squared distance, the p-median model gives a discrete version of the popular k-means model where the p (or k) centers must be on given data points (the k-means model seeks to find k center points in a given space, not necessarily on the given data points, that minimizes the sum of squared distances between the data points their closest center point). Fung and Mangasarian [74] showed that the

PMP could be combined with support vector machines to form a semi-supervised machine learning algorithm for labeling unlabeled data. The PMP is used in this context to choose the best pieces of data to have labeled by an expert and the support vector machine then uses that information to label the rest of the data.

Beyond facility location and data analysis, Briant and Naddef [24] used the PMP to solve the Optimal Diversity Management Problem. Many manufacturers, for example automotive manufacturers, will need many different configurations of a certain part corresponding to the different models and configurations of automobile that they produce. Unfortunately, the number of configurations of the part can be large enough to make it impractical to produce each configuration. For example, Briant and Naddef [24] state that some European manufacturers consider up to 7,000 different wiring designs for their cars. Some of these designs are capable of substituting for cheaper designs. Thus, in practice, only a small number, p, of different part configurations are produced. However, this requires an additional cost when a more expensive part configuration is used in place of a cheaper part configuration that is not produced. The Optimal Diversity Management Problem is to find the best p configurations to produce to minimize these extra costs. The connection to the PMP is clear. The part configurations are the demand points and the configurations actually produced are the chosen facilities in the PMP.

Because of its broad applicability, the PMP has been attacked with many different solution methods. Since the PMP is NP-hard [88], many of the methods used are approximation algorithms or heuristics; however, some researchers have used exact integer and linear programming methods. As previously mentioned, ReVelle and Swain [112] gave the most commonly used linear programming formulation for the PMP, which is the formulation that I use in this thesis. Avella

and Sassano [12] studied the p-median polytope and showed that the p-median problem could be modeled as a special case of the stable set problem on a specified auxiliary graph. This transformation uses the fact that no customer needs to be served by two different facilities. Therefore, from a graph perspective, no customer node will have two incoming edges in an optimal solution. Likewise, no node that is chosen to be a median will have an edge coming into it, and no node will have both an incoming edge and an outgoing edge since if it has an outgoing edge then it must be a median. Two edges that will never be in an optimal solution together are said to be dependent. The auxiliary graph defined by Avella and Sassano [12] has an vertex for each directed edge in the graph of the PMP instance and an edge between the vertices if the two edges of the PMP instance are dependent. Now, since a solution of the PMP will have $N - p$ edges where N is the number of vertices in the PMP instance, a stable set (or independent set) of size $N - p$ in the auxiliary graph is a solution to the PMP. The main focus of Avella and Sassano's [12] study was to find new facet-inducing inequalities rather than a new formulation. Therefore, although they performed some computational tests using a formulation based on the stable set perspective, they did not compare their formulation with the standard formulation given by ReVelle and Swain [112].

Cornuejols, Nemhauser, and Wolsey [43] developed another formulation (I will call it the CNW formulation) for the PMP using a canonical form for location problems. Instead of assigning each client to a specific vertex, this formulation tracks the distance from each client to any chosen facility vertex. Cornuejols, Nemhauser, and Wolsey [43] showed that the CNW formulation and its linear relaxation are equivalent to the standard formulation in the sense that their feasible regions are the same; however, their formulation can be smaller than the standard

formulation when the vertices are equidistant from multiple other vertices. Elloumi [60] improved on the CNW formulation by noting a recursive relationship between the variables used to track the distances from clients to chosen facilities. By altering the constraints to enforce this relationship, Elloumi [60] improved the CNW formulation to get a tighter formulation in the sense that the feasible region of the linear relaxation of Elloumi's formulation is strictly contained within the feasible region of the linear relaxation of the CNW formulation. However, even though Elloumi's [60] formulation is tighter than the CNW formulation, the optimal objective values of the two formulations (and of the standard formulation) are the same. Because the computational performance of the formulation is not the focus of this thesis, and they all have the same optimal objective value, I chose to use the simpler, more common standard formulation for this thesis.

One of the challenges to integer programming solutions to the PMP is the memory required to solve large instances. Consequently, there has been considerable research into solving larger and larger instances of the problem. Garfinkel, Neebe, and Rao [76] introduced the idea of column generation on a Set Partitioning formulation of the PMP, and this idea was further developed by du Merle, Villeneuve, Desrosiers, and Hansen [56]; and by Lorena, Senne, and Pereira [121] who proposed different ways of stabilizing the column generation approach to attain faster convergence. Lorena, Senne, and Pereira later incorporated their stablized column generation approach [121] into a branch-and-price method [122]. Avella, Sassano, and Vasil'ev [13] used a branch-cut-price approach together with the standard formulation to solve large problems. Their approach was very effective at increasing the size of problems that could be solved exactly; however, several of the moderately-sized test problems that they used remained intractable to their

approach. Thus, there is still a need for heuristic methods and approximation algorithms of the type developed in this thesis.

As with all well-known NP-hard problems, the PMP has been attacked with many different heuristic methods. These methods try to produce optimum or nearly optimum solutions in a practical amount of computational time. Unfortunately, to acheive practical running times, the heuristics must sacrifice guaranteed optimality, and they often have no guaranteed error bound. However, they remain useful because in many cases heuristics are the only practical methods for solving large problems. Mladenović, Brimberg, Hansen, and Moreno-Pérez [100] give a review of heuristic methods that have been applied to the PMP, and they divide the heuristics into four types: constructive heuristics, local search heuristics, mathematical programming heuristics, and metaheuristics. Although there is some overlap between these categories, I think that they provide a good framework for understanding the methods that have been applied to the PMP, and my explanation follows their framework.

Constructive heuristics build up a feasible solution from scratch. They do not need to start from some given initial feasible solution. The greedy algorithm is an example of a constructive heuristic. In a *greedy* algorithm for the p-median, the medians are chosen one-by-one until p medians have been chosen by choosing the median that optimizes some desired property. For example, Kuehn and Hamburger [91] developed a greedy algorithm for the PMP that at each step chooses the median that most reduces the total cost of supplying the demand points. A similar type of algorithm is the *greedy-drop* algorithm of Feldman, Lehrer, and Ray [69] which start with each demand point being a median and greedily removes medians until only p medians are left. Another type of constructive heuristic that has been

used for the p-median is the dual ascent heuristic developed by Erlenkotter [61] for uncapacitated facility location problems. Captivo [37] adapted the dual ascent strategy for the PMP.

Local search heuristics start from some given feasible solution and search for a better solution within the location of the current solution. For example, an early heuristic by Maranzana [95] starts from an arbitrary given set of p medians (for example, the output of a greedy algorithm) and assigns demand points to their closest median. Then for each set of demand points that is assigned to the same median, a 1-median problem is solved and the median is moved to the demand point that is the solution of the 1-median problem for that set of demand points. After choosing the new medians, all the demand points are again reassigned to their closest median, and the process repeats until no more changes are made. Another local search method is to start from some set of medians and move some median to a demand point that is not currently a median. If such a move reduces the total cost of the solution, then the current solution is kept and the process is repeated until no move can be found that decreases the solution cost. This method is called the *interchange* method and was introduced for the PMP by Teitz and Bart [128]. The error of local search methods for the p-median problem was investigated by Arya, Garg, Khandekar, Meyerson, Munagala, and Pandit [9]. Of course, the error of a local search method depends on the size of the location searched, and Arya et al. [9] showed that if the costs of the p-median instance are metric, and k facilities are allowed to be moved at once, then the cost of the local optimal solution found by the local search is at most $3 + \frac{2}{k}$ times the cost of the global optimum solution. To our knowledge, the current best approximation guarantee for a p-median algorithm is the $1 + \sqrt{3} + \epsilon$ bound of Li and Svensson [93]. The branch

decomposition algorithm presented in this thesis can be thought of as a local search heuristic since it essentially finds the best solution within a location given by some input heuristic or linear programming solutions.

Mathematical programming heuristics solve a reduction or relaxation of the PMP. For example, Baker [15] and Hribar and Daskin [85] gave dynamic programming schemes for the p-median that limit the state space of the dynamic program. Hribar and Daskin's [85] scheme builds up an i-median solution out of (i-1)-median solutions at each step. Their heuristic reduces computational effort by only storing some specified number, H, of i-median solutions at each step. Hribar and Daskin [85] note that that their heuristic is identical to greedy if $H = 1$ and is an exact algorithm if $H > \max_{0 \leq j \leq P} \binom{N}{j}$. Baker's [15] scheme allows the state space to increase linearly with problem size.

Other mathematical programming heuristics solve a relaxation of some formulation of the PMP. Such heuristics use a Lagrangean relaxation of some constraint in the formulation. The resulting relaxation solution gives a lower bound on the cost of an optimal solution and some adjustment can be applied to the relaxation solution to give a feasible heuristic solution. For example, a common relaxation used by Cornuejols, Fisher, and Nemhauser [42]; Narula, Ogbu, and Samuelsson [103]; and Beasley [21] relaxes the constraint requiring each facility to receive exactly 1 unit of supply. Since the formulation still requires p medians to be chosen, it is easy to find a feasible solution by assigning each demand point to its closest open median.

The last type of heuristics are metaheuristics. The bulk of recent work on the PMP falls into the metaheuristic category. The goal of metaheuristics is to find a close to optimal solution; however, they generally do not provide any information

about the error of their solutions. Without running a metaheuristic, there is no theory for determining how far the solution provided by the metaheuristic will be from optimal. Also, metaheuristics contain a random element to escape from local optima and running the algorithms multiple times may give multiple distinct solutions. Consequently, these methods are often run multiple times and the best solution from the multiple runs is used. For a survey of metaheuristic approaches that have been applied to the PMP, see Mladenović, Brimberg, Hansen, and Moreno-Pérez [100]. For the purposes of this thesis, the most important metaheuristics are the genetic algorithms, the hybrid heuristic of Resende and Werneck [111], and the heuristic concentration methods.

The genetic algorithms are based on the idea of genetic drift, in which a living population reacting to selection pressure tends to produce children that are better adapted to the selection pressure. The idea of genetic algorithms is to model solutions to the PMP (or whatever problem is being solved) as chromosomes and have those solutions breed together to produce offspring solutions while selecting for the lowest cost solutions. A random mutation is also introduced to the offspring in an attempt to mimic the mutations seen in natural populations and allow better solutions to be produced even if the initial population is low quality. As more generations of chromosomes are produced, the population is expected to drift towards the optimum solution. The output of the genetic algorithm is the best solution found after some number of generations. Hosage and Goodchild [84] were the first to apply the genetic algorithm framework to the PMP; however, they did not require their solutions to have exactly p medians. Instead, they penalized the objective for solutions that had the wrong number of medians. Unfortunately, as Correa, Steiner, Freitas, and Carnieri [44] noted, allowing solutions to have the

wrong number of medians greatly increases the number of solutions that may be in the population, and time can be wasted generating solutions that are infeasible. Dibble and Densham [52] gave a better encoding of p-median solutions that requires solutions to have exactly p medians, and their encoding has been used by subsequent authors. Bozkaya, Zhang, and Erkut [23] and Alp, Erkut, and Drezner [5] made improvements to the way that offspring are produced. Correa et al. [44] proposed adding a new type of mutation, which they called a hypermutation. This hypermutation is basically the application of a local search heuristic to improve each element of the current population. Correa et al. [44] used an interchange heuristic similar to Teitz and Bart's [128] heuristic. Their computational experiments showed that the addition of the hypermutation allowed their genetic algorithm to outperform their implementation of the tabu search heuristic for the capacitated PMP. Recently, Rebreyend, Lemarchand, and Euler [108] created an improved version of Correa et al.'s [44] method, which they called *imp-GA*. Rebreyend et al.'s [108] adaption was to limit the size of the local search by just interchanging some small number of the chosen facilities instead of all of them. Their computational results show that this adaption allows imp-GA to perform better on instances of the PMP with at least 1000 demand points. Because of its performance on large instances of the PMP, imp-GA is one of the algorithms to which I compare my branch decomposition algorithm in this thesis.

Another algorithm that I use for comparison in this thesis is the hybrid heuristic of Resende and Werneck [111]. This algorithm is an improvement built onto the GRASP heuristic of Feo and Resende [70]. GRASP stands for Greedy Randomized Adaptive Search Procedure, a description of this procedure for generic problems was given by Feo and Resende [71]. At a high level, a GRASP consists of a randomized

greedy construction of a feasible solution that is then passed to a local search method that improves the greedily constructed solution. The randomization comes during the greedy construction of a solution. In a GRASP, the best candidate is not necessarily chosen as it would be in a pure greedy method. Instead, a candidate is chosen randomly from a list of the best candidates available. The function determining the value of the candidates can be changed as the greedy construction progresses and provides the adaptive part of the procedure.

Resende and Werneck [111] applied the GRASP heuristic to the PMP with an additional path-relinking step and an additional population generation phase that creates a new population of solutions by combining the best solutions from the previous iteration, similar to a genetic algorithm. GRASP with path-relinking has since been applied to different variants of the PMP. For example, Arroyo, Soares, and dos Santos [8] applied it to the bi-objective PMP, and Pérez, Almeida, and Moreno-Vega [105] applied it to the capacitated PMP. According to Avella, Boccia, Salerno, and Vasil'ev [10]; Avella, Sassano, and Vasil'ev [13]; Mladenović et al. [100]; and Sáez-Aguado and Trandafir [120], Resende and Werneck's [111] hybrid heuristic has been one of the most effective algorithms for the PMP. Consequently, I use it to evaluate the algorithm presented in this thesis.

The algorithm presented in this thesis belongs to the class of heuristic concentration algorithms. The unifying idea of these algorithms is to create a pool of solutions from different heuristics or from multiple runs of a single heuristic, and to use this pool of solutions to inform the creation of a new, hopefully improved, solution. Heuristic concentration was introduced by Rosing and ReVelle [115]. These authors used multiple runs of Teitz and Bart's [128] interchange heuristic to build up a list of heuristic solutions. Then, they built a *concentration set* of

locations where a facility was assigned by at least a threshold percentage of the heuristic runs. Finally, they used integer programming with CPLEX to find the best solution to the reduced problem where facilities must be assigned to locations from the concentration set. Rosing, ReVelle, Rolland, Schilling, and Current [116] compared Rosing and ReVelle's [115] heuristic concentration method to Rolland, Schilling, and Current's [114] tabu search heuristic. They found that the heuristic concentration method was superior in terms of solution quality. In terms of computational time, the heuristic concentration method was faster on some instances and the tabu search was faster on others.

The difference between the method presented in this thesis and the previous heuristic concentration methods is in both the method used to solve the reduced problem and in the number and type of heuristic runs that are used to build the concentration set. Exact methods of solving the reduced problem have used integer programming. Serra, ReVelle, and Rosing [123] used an improved version of Teitz and Bart's [128] heuristic created by Densham and Rushton [51] to both create the concentration set and heuristically solve the reduced problem. Rosing, ReVelle, and Schilling [117] use a 2-opt procedure. In contrast, the method presented here uses a branch decomposition based dynamic programming algorithm to solve the reduced problem. In addition, Rosing and ReVelle's [115] initial paper used 200 runs of Teitz and Bart's [128] interchange heuristic to create the concentration set. One of the questions with which Rosing and ReVelle [115] ended their paper was whether such a large number of heuristic runs was necessary. This thesis uses a much smaller number of runs and still acheives improvements over the concentration set. Finally, instead of the interchange heuristic, this thesis shows that the linear programming relaxation and Resende and Werneck's [111] GRASP heuristic also provide good

concentration sets.

Chapter 3

Solving the p-Median Problem with Branch Decompositions*

3.1 Introduction

Solution approaches to NP-hard problems fall into four basic categories: meta-heuristics, combinatorial approximation algorithms, exact combinatorial algorithms, or integer programming. Since the problems are NP-hard, exact solution methods are often impractical. On the other hand, sub-optimal solutions such as those provided by meta-heuristics and approximation algorithms require practicioners to accept costs that could be avoided with a better solution.

In this chapter, I use a branch decomposition technique to improve approximations to the p-median problem. This technique can be used to develop a solution from the linear relaxation of the problem, or to combine a pool of heuristic solutions into a solution of higher quality than any in the original pool. My computational results show that my technique provides solutions of better quality than popular heuristics, while being significantly faster than integer programming on graphs with a low width branch decomposition.

The p-Median Problem (PMP) was introduced in Chapter 1.2. Although it is NP-hard in general [88], ReVelle and Swain [112] observed that their linear programming formulation of the PMP often gave integer solutions. Since linear

*This chapter is adapted from [68].

programming is solvable in polynomial time (for example by Khachiyan's ellipsoid method [90] or Karmarkar's interior point method [89]), this observation leads to the question of whether linear programming can be used in practice to solve the PMP. Unfortunately, as problem sizes get larger, fractional solutions to the linear program become more common. However, in many cases these fractional solutions have simple structure. For example, half-integral fractional solutions are common. Some preliminary computational experience showed that the simple structure of the linear programming solutions translated to low branchwidth. Consequently, the PMP is a prime target for a branch decomposition algorithm.

In this chapter, I develop a heuristic for the PMP using branch decompositions of support graphs produced either by heuristics or by linear programming. Section 3.2 gives the integer programming formulation of the PMP. Section 3.3 introduces my algorithm. I show complexity results and theoretical error bounds in Section 3.4. I give computational results comparing my algorithm to other state-of-the-art algorithms in Section 3.5, and I offer conclusions in Section 3.6.

3.2 Preliminaries

I will consider the PMP defined on a complete directed graph (that is, a graph where directed edges exist in both directions between each pair of nodes), call it $\vec{K}(V, A)$, with edge costs c. The problem asks for a subset $M \subseteq V$ such that $|M| = p$ and the sum of the distances from each node of $V \backslash M$ to it's closest neighbor in M is minimized. Definition 3.1 gives the standard, straightforward formulation for the PMP. Alternative formulations exist for the problem. Avella and Sassano [12] gave a formulation that relates the PMP to the stable set problem. Elloumi [60] gave a formulation in terms of neighborhoods, which requires less linear constraints if the

distance matrix is sparse. However, since I deal with complete graphs in this chapter, the distance matrix is dense, and I use the standard formulation.

Definition 3.1 Integer Program Model of the PMP (PMPIP)

minimize $c^T x$

subject to:

$$\sum_{v \in V} x_{v,v} = p \qquad (1)$$

$$\sum_{v_i \in V} x_{v,v_i} = 1 \qquad \forall v \in V \qquad (2)$$

$$x_{v_1,v_2} \leq x_{v_2,v_2} \qquad \forall v_1, v_2 \in V \qquad (3)$$

$$0 \leq x_{v_1,v_2} \leq 1 \qquad \forall v_1, v_2 \in V,\ v_1 \neq v_2 \qquad (4)$$

$$x_{v_1,v_2} \in \{0,1\} \qquad \forall v_1, v_2 \in V,\ v_1 = v_2 \qquad (5)$$

The linear relaxation of an integer program is the linear program that arises when the integrality constraints of an IP (the constraints (5) in the PMPIP) are relaxed to allow fractional values. I wish to solve the linear relaxation of the PMP integer program (PMPIP) and exploit this solution to find low-cost feasible solutions to the PMPIP. I will call this relaxation the PMPLP. In particular, I am interested in the *support graphs* of the relaxations. In my case, since I posed the PMP over a complete graph, where the nodes are demand/median locations, the *support graph* of the PMPLP solution contains the nodes of the complete graph together with all edges whose corresponding variables take on positive value in the solution of the PMPLP. All the edges whose corresponding variables have a value of 0 in the PMPLP solution are not in the support graph.

Note that in considering the support graph of the PMPLP, I simply use the edges whose corresponding variables exceed a threshold value of 0.0001 in the PMPLP solution. I do not create any new edges by shortcutting shortest paths (shortcutting means adding a new edge between two nodes with cost equal to the

cost of the minimum cost path between the two nodes). Thus, if an edge variable is not positive in the PMPLP, then that edge is not in the PMPLP support graph, and the two ends of that edge cannot be linked in a PMP solution contained in the PMPLP support graph. Since many of the edges present in the complete graph may not be present in the PMPLP support graph, it is possible that for every potential subset of nodes, M, with $|M| = p$, there exists a node, $v \in V \backslash M$, such that none of the edges directly connecting v to a node in M are present in the PMPLP support graph. That is, it is possible that no dominating set of size p exists in the PMPLP support graph. In this case, for any choice of p medians, there will be some node that cannot be linked to a median using an edge of the PMPLP support graph. Thus, the support graph of the PMPLP will not contain a solution of the PMP, and some alteration of the PMPLP will be required. I explore three ways to alter the PMPLP in Section 3.3.1.

In certain cases, for example when the PMP instance is too large to be solved by current linear programming software or if the linear program is too slow, it is necessary to solve the PMP using heuristics. Although these heuristic solutions may not be optimal, I expect that a combination of these solutions will contain information that will help us build a better solution than any of the individual solutions. In a similar manner to the PMPLP, I can build a support graph out of multiple heuristics by simply including any edge of the complete graph that is used in at least one of the heuristic solutions.

Whether I use the PMPLP or heuristics to create a support graph, I will use dynamic programming to mine the useful information contained in it. While Hribar and Daskin [85] have previously developed a dynamic programming heuristic for the PMP, my method is unique because I use branch decompositions to form the

dynamic programming subproblems. See Section 1.4 for further explanantion of branch decompositions.

3.3 The Branch Decomposition Heuristic

Dynamic programming algorithms have two essential components: a decomposition of the problem into subproblems and a method for using smaller subproblem solutions to build solutions for larger subproblems. In this section, I describe how I use branch decompositions to build these components and incorporate them into my algorithm.

In my algorithm, I consider the directionality of the edges when I solve the smallest subproblems in the dynamic programming. Thus, rather than using branch decompositions of directed graphs I only consider the branch decompositions of the underlying undirected graphs. In theory, this distinction makes little difference because, given a directed graph, the branch decomposition of the underlying undirected graph can be extended to a branch decomposition of the directed graph by simply adding a pair of leaves representing each direction descending from the leaf representing the undirected edge. This extension does not affect the width of the decomposition.

For my heuristic, given a support graph, G, that comes from either linear programming or heuristics, and a branch decomposition of G, I start by defining the subproblems for dynamic programming. I have a subproblem for each vertex of the branch decomposition tree, and for a given tree vertex, the corresponding subproblem is a p-median problem on the subgraph induced by the leaves descending from that tree vertex. For leaf vertices of the branch decomposition tree, these subproblems are p-median problems on single edges of G. For my algorithm, I

must determine how the allowed p medians will be distributed on the subgraphs. It is likely that, in an optimal solution, the medians will be spread throughout the graph, and the subgraphs will only use a fraction of the medians. However, I know of no theory that will allow me to distribute the medians to the branches of the decomposition while guaranteeing that solution quality will not suffer. Therefore, I simply allow each branch of the decomposition to use the full allotment of p medians. I ensure that no more than p medians are used in the final solution by only merging partial solutions that together have no more than p medians.

Since all p medians are allowed on each branch, there are six possible solutions to the subproblems on the leaves of the decomposition, see Figure 3.1. Each end of the corresponding edge can either be a median, be linked to the other end of the edge, or it can be free. Free nodes are necessary because, in a solution of the full problem, it is possible that a node may be linked to a median that is not part of the current subproblem, and if a node is either a median or linked to a median in a subproblem it will never be linked to a median outside of the subproblem. However, if one of the nodes incident to the edge corresponding to a leaf vertex has a degree of 1 in G, then that node will not be part of the middle set of any branch decomposition vertex and cannot be free. For such edges, there are at most the 4 possible solutions where the node with degree 1 is not free.

Once I have solutions for the leaves of the decomposition, I form solutions to the interior vertex subproblems by merging a partial solution from one of the children of the interior vertex with a partial solution from the other child of the vertex. The operation of merging solutions forms the solution of the new subproblem by scanning through each of the child subproblems and setting each node and edge in the new subproblem to have the same value as in one of the child subproblems.

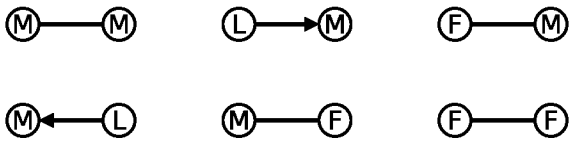


Figure 3.1 : Dynamic programming base cases for leaves of decomposition. M denotes a median node, L denotes a node linked to a median, and F denotes a free node.

Thus, a node that is linked to a median in a child subproblem remains linked, a node that is a median remains a median, etc. Nodes that are not in either child subproblem are also not part of the subproblem on the new vertex. It is obvious that this process leads to conflicts if the two child subproblems disagree on the state of a node, i.e. one child has a node linked to certain median and the other child has the node linked to a different median. I call these partial solutions *incompatible*.

My goal with defining compatible solutions is to reduce the number of partial solutions that I need to merge in the dynamic programming stage. Since I do not need to merge an incompatible pair of solutions, it is in my best interest to make as many pairs incompatible as possible while still retaining an optimal solution. I say that two states of a middle set node are compatible if a partial solution with one state at the node can be merged with a partial solution that has the other (not necessarily different) state on that node without violating some property of integral p-median solutions. I say that two partial solutions are compatible if each of the nodes in the intersection of their middle sets have compatible states. The compatibilities of the three states for middle set nodes are given in Table 3.1.

Most of the definitions in Table 3.1 are straightforward; however, I will still take the time to explain them here. A median state is compatible with a median state

Table 3.1 : Compatibility for middle set nodes

	Median	Linked	Free
Median	Compatible	Incompatible	Incompatible
Linked	Incompatible	Incompatible	Compatible
Free	Incompatible	Compatible	Compatible if in middle set

because both partial solutions agree on the state of the node, thus the solutions can be merged without a problem. However, the median state is not compatible with the linked state because a merged solution would have that node being both a median and linked to another median, which is unnecessary because a median node is considered to satisfy its own demand. Median nodes are also defined to be incompatible with free nodes. I chose to make these states incompatible to reduce the number of compatible solutions. This definition is safe because if a middle set node is a median in one partial solution (L) and free in the other (R), then it will be a median in the merged solution. Thus, I get the same merged solution as I would have by merging L with the partial solution (P) obtained by changing the node to a median in R. P must be one of the partial solutions of the R child branch because if L and R could be merged without exceeding p medians, then there are no more than p medians in P. Thus, P must be in the list of partial solutions for the R child branch. It follows that I am not discarding any possible solutions by declaring the median state to be incompatible with the free state.

We saw that the linked state was incompatible with the median state. The linked state is also incompatible with the linked state. If a middle set node is linked to a node that is not in the middle set in one of the partial solutions, then in the

merged solution, it will be linked to two different nodes, which is unnecessary because linking to one median is all that is needed to satisfy a node's demand. If the node is linked to the same middle set node in both partial solutions, then I could have just used one partial solution where the node was free. Thus, the linked state is incompatible with the linked state. However, the linked state is compatible with the free state, because in that case the node is linked to a single median in the merged solution.

The only pair left to determine is the compatibility of the free state with the free state. If the node is not just in the middle set of the child branches but is also in the middle set of the current branch decomposition vertex, then we must have that the free state is compatible with the free state. However, nodes cannot be free in the final solution. So, if the node is in the middle set of the child vertices but not in the middle set of the current branch decomposition vertex, then the free state is not compatible with the free state.

Now that I have a rule for determining compatibility, I build partial solutions for a branch decomposition vertex by merging compatible partial solution pairs from its two children. However, when building solutions in this manner, it is possible that two partial solutions, A and B, will have the same number of medians and the same middle set configuration, but different configurations on the nodes that are not in the middle set. A and B will both be compatible with the exact same set of solutions from the other branch decomposition vertices. However, if A costs less than B, then A will give the lower cost in the merged solutions. Thus, I only need to store the lowest cost partial solution with each middle set configuration and number

of medians. I store the partial solutions in a hash table with the hashing function

$$f(x_0, x_1, \ldots, x_{K-1}, m) = \sum_{i=0}^{K-1} 3^i x_i + 3^K m$$

where $x_i \in \{0, 1, 2\}$ represent the state of the middle set node i, K is the width of the branch decomposition, and m is the number of medians used.

Since I am building solutions by merging partial solutions from the children, I must first obtain the partial solutions on the children. A post-depth-first search ordering ensures that the children will be processed before the parent. When I finally reach the root vertex of the branch decomposition, the partial solutions become complete, and I choose the lowest cost solution at the root vertex to be the solution returned by my algorithm. The dynamic programming procedure is summarized in Algorithm 3.1.

3.3.1 Performance Tweaks

In my numerical experiments, and in practical application, situations arise where Algorithm 3.1 needs to be tweaked somewhat. First, as explained in Section 3.2, it is possible that no feasible solution exists for the given p on the PMPLP support graph. Since my branch decomposition algorithm on the PMPLP support (BDPM-LP) finds solutions only on the PMPLP support graph, my algorithm may not be able to find a feasible solution. There are three ways to deal with this problem. The first way is to shortcut the graph. This method essentially entails adding edges to the support graph until a feasible solution can be found. Unfortunately, adding edges to the graph can increase its branchwidth, and since the complexity of my algorithm is so closely tied to branchwidth, shortcutting will likely have a detrimental effect on efficiency.

Algorithm 3.1: Dynamic Programming on Branch Decompositions

Data: post-DFS ordering of a branch decomposition of G

Result: Solution of p-Median problem on G

Initialize leaves of branch decomposition with their possible solutions;

for *branch decomposition vertices in post-DFS order* **do**

  for *i a solution of left child* **do**

    for *j a solution of right child* **do**

      if *(i and j are compatible)* **then**

        Store the compatible pair (i,j);

      end

    end

  end

  for *each compatible pair (i,j)* **do**

    merge i and j to get new solution;

    if *(cost of new solution < cost of solution in hash table)* **then**

      Store new solution in hash table;

    end

  end

end

Output least cost solution of root vertex;

The second way to deal with the problem, is to add a cutting plane to the PMPLP. If no integral solution exists on the support graph of the PMPLP, then any integral solution must contain an edge that is not in the support graph. Let E be the set of edges in the support graph of the PMPLP. Then the constraint

$$\sum_{(i,j)\notin E} x_{i,j} \geq 1$$

is a valid cutting plane for the PMPIP. Adding these cutting planes will eventually force an integral solution to exist on the support graph of the linear program. Unfortunately, this cutting plane does not seem to be facet-inducing. In my preliminary experiments, this method did not perform well because each cutting plane did not cut off enough of the feasible region. Also, this method tended to increase the branchwidth of the support graph.

The third method, the one that I use in my computational experiments, is what I will call fixing. *Fixing* entails picking some node variable, $x_{i,i}$, that takes positive, fractional weight in the PMPLP, and adding the constraint $x_{i,i} = 1$. The PMPLP is then resolved with this added constraint, and I run my algorithm on the support graph of this new solution. Although this approach prevents the method from finding solutions where the fixed node is not a median, it does eventually force a feasible solution to exist on the PMPLP support graph.

Because the performance of BDPM-LP depends on the branchwidth of the PMPLP support, I want to avoid runnning the heuristic on graphs with high branchwidth. Fixing a single variable will not necessarily decrease the branchwidth and may in fact increase it. However, repeatedly fixing variables will eventually reduce the branchwidth of the support graph. This statement can easily be seen from the fact that fixing will eventually cause the PMPLP solution to be integral.

Consequently, I can limit the branchwidths of the support graphs that I use for BDPM-LP by fixing whenever the width of my branch decomposition is above some given limit. In my numerical experiments, my heuristic became impractical with widths larger than 7. Consequently, I tweaked BDPM-LP by fixing whenever the width of the branch decomposition was above 7. In a similar manner, the algorithm using multiple heuristic runs (BDPM-H) can have widths that are too high. I correct this by using fewer heuristics to create the support graph.

There are multiple fixing strategies, such as fixing lexicographically or fixing the node with largest fractional weight. In my numerical experiments, I use lexicographic fixing. Given some indexing of the nodes, I simply scan the list of node variable values from the LP solution and fix the fractional node with the lowest index.

3.4 Complexity and Error Bounds

In this section, I give a complexity result and a theoretical error bound for BDPM-LP. I do not give similar error bounds for the version of my algorithm where the support graph comes from heuristics (BDPM-H) because the heuristics themselves generally do not have error bounds. In general, all that I can say about the error of BDPM-H is that it is at least as good as the heuristics used to build the support graph.

Theorem 3.1

Let K be the width of a branch decomposition of a graph with N vertices and M edges. Then, Algorithm 3.1 using the given decomposition requires $\mathcal{O}(2M(N + p^2 9^K(K+1) + p^2 5^K N))$ *operations.*

Proof: The branch decomposition has $2M - 2$, or $\mathcal{O}(2M)$, vertices, and at each vertex the algorithm needs to find the pairs of compatible solutions and merge them. To do this, I first need to find the middle set of the branch decomposition vertex. This step requires checking the induced degree of each node against its actual degree. Thus, the step requires N comparisons. Now, each node in a middle set can have at most three configurations. Thus, each vertex of the decomposition has at most $p3^K$ partial solutions.

To find the compatible solutions, the algorithm compares the configuration of each node in the middle set for each pair of partial solutions, one from each child vertex in the decomposition. Since the number of nodes in the middle set is limited by K, this step uses at most $p3^K p3^K(K+1) = p^2 9^K(K+1)$ comparisons. Finally, the compatible solutions must be merged. Since all configurations are not compatible with each other, there are a maximum of $p^2 5^K$ compatible pairs. The solutions are merged by simply scanning through each of the partial solutions and setting the new solution to its values in the partial solutions. Since there are at most N nodes that must be scanned, merging the solutions requires $p^2 5^K N$ steps. Thus, the complexity of the algorithm is $\mathcal{O}(2M(N + p^2 9^K(K+1) + p^2 5^K N))$. ■

The fixing strategy introduced in Section 3.3.1 complicates the development of theoretical worst-case error bounds for BDPM-LP because the optimal cost of the linear program increases when fixing occurs, but if the fixing step is not necessary, then I can provide a theoretical bound.

Theorem 3.2

Let $\bar{x}$ be the optimal solution to the PMPLP, and let α be the smallest integer such that $\alpha\bar{x}$ is integral. Let $\bar{h}$ be the solution returned by algorithm 3.1, and let c be the cost vector for the instance. If no fixing is required, then $c^T\bar{h} \leq \alpha c^T\bar{x}$.

Proof: Since $\alpha\bar{x}$ is integral, it contains at least one copy of each edge in the PMPLP support graph. Algorithm 3.1 returns a solutions whose edges are limited to the PMPLP support, and this solution has at most one copy of each of these edges. So, $\alpha\bar{x}$ has at least as many copies of each edge as are contained in $\bar{h}$. Thus, $c^T\bar{h} \leq \alpha c^T\bar{x}$. ■

For the interesting case of half-integral solutions, Theorem 3.2 gives an error bound of 2 when fixing is not needed. Note that on these limited instances, my bound is better than the $1 + \sqrt{3} + \epsilon$ bound of Li and Svensson [93], but in general it is worse. However, in my computational experiments, whether fixing was needed or not, the average performance of my algorithm is much better than the worst-case bound.

3.5 Computational Experiments

In this section, I report computational results for both the PMPLP-based algorithm and the heuristic-based algorithm. The first strategy for which I report computational results is the use of multiple runs of the GRASP heuristic of Feo and Resende [71] to create the support graph. In my experiments, I use four runs of GRASP to create the support graph on which I run BDPM. An edge is in this support graph if and only if it is in one or more of the four GRASP solutions. I call this heuristic BDPM-GRASP. The second strategy for which I report results is BDPM-LP. In this method, an edge is in the support graph if and only if its corresponding variable has a value at least 0.0001 in the linear programming solution. I compare the results of my algorithm to GRASP, integer programming, the imp-GA algorithm of Rebreyend, Lemarchand, and Euler[108], and the hybrid heuristic (HHP) of Resende and Werneck [111]. The imp-GA algorithm was

demonstrated by Rebreyend et al. [108] to outperform other genetic algorithms on large problems, and HHP is also known to be state-of-the-art for p-median problems (see for example Avella et al. [10] or Mladenović et al. [100]).

My computational results were obtained on a Dell Precision T1650 workstation with a 3.3 GHz Intel Core i3-2120 CPU, 3.7 GB of RAM, and Red Hat Enterprise Linux version 6.6. The code was written in C++ and compiled with g++ version 4.4.7. Integer and linear programs were solved using Gurobi version 5.5.0 with the dual simplex method and a single thread. Branch decompositions were found using the C++ code of Hicks [80]. GRASP results and HHP results were obtained from the POPSTAR code of Resende and Werneck [111]. I obtained imp-GA results from my own implementation in C++.

The test instances that I use in this section come from the TSPLIB [110] or the ORLIB [20]. These two libraries are the standard test instances for the PMP (for example, see Mladenović et al. [109]). From the TSPLIB, I used lin318, rd400, si535, ali535, rat575, gr666, u724, dsj1000, pr1002, rl1304, nrw1379, fl1400, u1432, vm1748, d2103, and pcb3038. Many of these instances have been used before as test instances for the p-median problem, see for example Avella et al. [13] and Garcia et al. [75]. For each file chosen from the TSPLIB, I ran instances with the number of medians, p, in the set 5, 10, 50, 100, 200, 300, 400, 500, up to half the number of vertices in the instance. In keeping with standard practice in recent papers on p-median, see for example [13], [60], and [75], I only report computational results for instances pmed26-pmed40 from the ORLIB.

The running times of both BDPM versions show a loose correlation with the width of the branch decomposition used, which should be expected given Theorem 3.1. However, since it is possible for the decomposition to have only one large

middle set and for the rest of the middle sets to be small, or for a linear program solution to be mostly integer with only a small fractional part with high width, higher widths do not necessarily lead to higher running times.

For both the TSPLIB instances and the OR-Library instances, BDPM-LP is generally faster and can solve larger problems than integer programming. It is also more accurate than HHP when fixing is not required. BDPM-LP was faster than imp-GA and more accurate than imp-GA when $p > 50$. If the time to solve the linear program is not counted, then the running times of BDPM-LP were competitive with HHP for low widths. Also, BDPM-GRASP was able to improve on the best GRASP run; was faster than imp-GA; and was more accurate than imp-GA for $p > 100$. The following subsections give more detailed discussion of these results.

3.5.1 BDPM-GRASP

For BDPM-GRASP, I use four runs of GRASP to form the support graph for my algorithm. For some instances, the width of the decomposition of the support graph using four runs was too high for BDPM-GRASP to be practical. If BDPM-GRASP was unable to solve an instance, I simply decreased the number of GRASP runs that I use to build the support graph. Detailed results for these difficult instances, including the number of runs required, are reported in Table 3.9. I compare the results of BDPM-GRASP to integer programming, imp-GA, HHP, and to the best GRASP run from the four used to build the support graph. Average results for each of these methods are reported in Table 3.2. The averaged results in this table show that BDPM-GRASP slightly reduced the error of the best GRASP run, and BDPM-GRASP only had smaller error than imp-GA on the ORLIB instances. However, the averaging hides a clear trend in my data. My data shows that

BDPM-GRASP performs better as p increases, and imp-GA performs worse. I consistently saw improvement over the best GRASP run when p was at least 50. Likewise, BDPM-GRASP consistently had less error than imp-GA when p was at least 200 for small instances and 100 for large instances. These trends are seen in Tables 3.3 and 3.4.

For the OR-Library instances, the best GRASP run was often exact. However, on average over all the OR-library instances, the average error ratio (calculated as the average of the heuristic cost divided by the true solution cost for each instance) of BDPM-GRASP was 1.00018 (this error ratio was calculated using the best GRASP solution for the two instances pmed37 and pmed40 where the width was too high for BDPM-GRASP), while the best of the GRASP runs had an average error ratio of 1.00153. Imp-GA had an average error ratio of 1.00136. On average BDPM-GRASP removed 32.9% of the error of the best GRASP run (calculated as the average of the percentage of the GRASP error that was still present in BDPM-GRASP for each instance). Thus, on these instances, BDPM-GRASP was more accurate than both imp-GA and GRASP. Detailed results for these instances are reported in Table 3.5.

For the TSPLIB instances, I divide the instances into two groups. The small instances have less than 1000 vertices, and the large instances have at least 1000 vertices. The average error ratio of my algorithm for small instances was 1.0373 compared to an average error ratio of 1.0392 for the best GRASP run and 1.004 for imp-GA. On average for small instances, BDPM-GRASP removed 27.8% of the error of the best GRASP run. Detailed results for small instances are reported in Table 3.6. The average error ratio of BDPM-GRASP for large instances was 1.0353 compared to an average error ratio of 1.0388 for the best GRASP run. On average

for large instances, BDPM-GRASP removed 37.0% of the error of the best GRASP run. Detailed results for large instances are reported in Tables 3.7 and 3.8.

Although the average results show a small improvement from using BDPM-GRASP over GRASP, and no improvement from using BDPM-GRASP over imp-GA, the benefit of using BDPM-GRASP becomes apparent when results are broken down according to the p values in the instances. Tables 3.3 and 3.4 and Figures 3.2 and 3.3 show average relative errors for different p values, and it is clear from Figures 3.2 and 3.3 that as p increases, the average BDPM-GRASP solution improves relative to the GRASP solution. Tables 3.3 and 3.4 indicate that BDPM-GRASP offers significant improvement over GRASP when p is at least 50. For the small instances, Table 3.3 shows that BDPM-GRASP outperforms imp-GA when p is 200 or 300, and for the large instances, Table 3.4 shows that BDPM-GRASP outperforms imp-GA when p is at least 100.

Except for a few instances, the HHP algorithm is slightly more accurate than BDPM-GRASP. However, HHP is slower than BDPM-GRASP when branchwidths are less than 6. This result is expected since HHP is also an improvement built on top of GRASP, but HHP explores a larger search space while attempting to improve the GRASP solution. Also, since HHP uses different methods than BDPM, the two methods could potentially be combined to form a new method for the p-median problem.

Table 3.2 : Average results for BDPM-GRASP.

Algorithm	ORLIB (15 instances)			Small TSP (36 instances)			Large TSP (64 instances)		
	# solved	time	error	# solved	time	error	# solved	time	error
BDPM-GRASP	13	45.3	1.0002	36	22.19	1.037	69	65.7	1.032
Best GRASP	15	0.03	1.0015	36	0.025	1.039	70	0.14	1.036
imp-GA	15	730	1.0014	36	551.3	1.004	70	8130	1.024
IP	12	998.6	1.0	36	270.6	1.0	29	162.1	1.0
HHP	15	1.37	1.0	36	1.21	1.037	70	10.15	1.030

Note: The # solved column indicates the number of instances that the relevant algorithm solved without exhausting available memory. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required to run GRASP.

Table 3.3 : Average relative errors for BDPM-GRASP on small TSPLIB instances broken down by p.

Algorithm	p=5	p=10	p=50	p=100	p=200	p=300
BDPM-GRASP	0.12	0.0519	0.0107	0.0052	0.0021	0.00035
Best GRASP	0.12	0.0520	0.0126	0.0073	0.0059	0.0085
imp-GA	**0.00054**	**0.0022**	**0.0039**	**0.0044**	0.0063	0.0145
HHP	0.122	0.0517	0.0087	0.0058	**0.0013**	**0.0003**

Note: BDPM-GRASP performs better as p increases, while imp-GA performs worse as p increases. Bold text indicates the method that had the least error.

Table 3.4 : Average relative errors for BDPM-GRASP on large TSPLIB instances broken down by p.

Algorithm	p=5	p=10	p=50	p=100	p=200	p=300	p=400	p=500
BDPM-GRASP	0.1911	0.0623	0.0131	0.0079	0.0043	0.0053	0.0049	0.0041
Best GRASP	0.1911	0.0623	0.0137	0.0105	0.0084	0.0106	0.0117	0.0103
imp-GA	**0.0007**	**0.0009**	**0.0070**	0.0087	0.0211	0.0409	0.0544	0.0536
HHP	0.1911	0.0622	0.0101	**0.0038**	**0.0020**	**0.0018**	**0.0017**	**0.0026**

Note: BDPM-GRASP performs better as p increases, while imp-GA performs worse as p increases. Bold text indicates the method that had the least error.

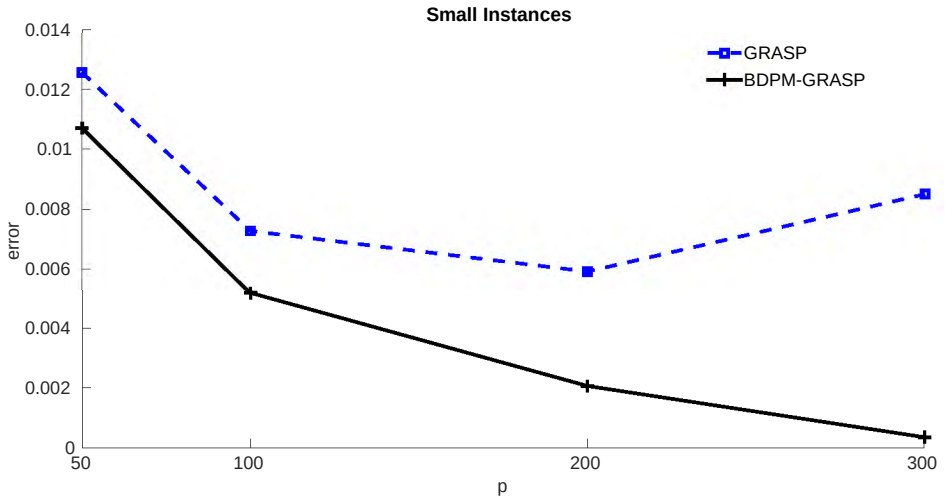


Figure 3.2 : Comparison of BDPM-GRASP to GRASP for small TSPLIB instances. This plot compares average relative error for BDPM-GRASP (solid line, +) and the best GRASP run (dashed line, □) for different values of p and test instances from the TSPLIB with less than 1000 vertices.

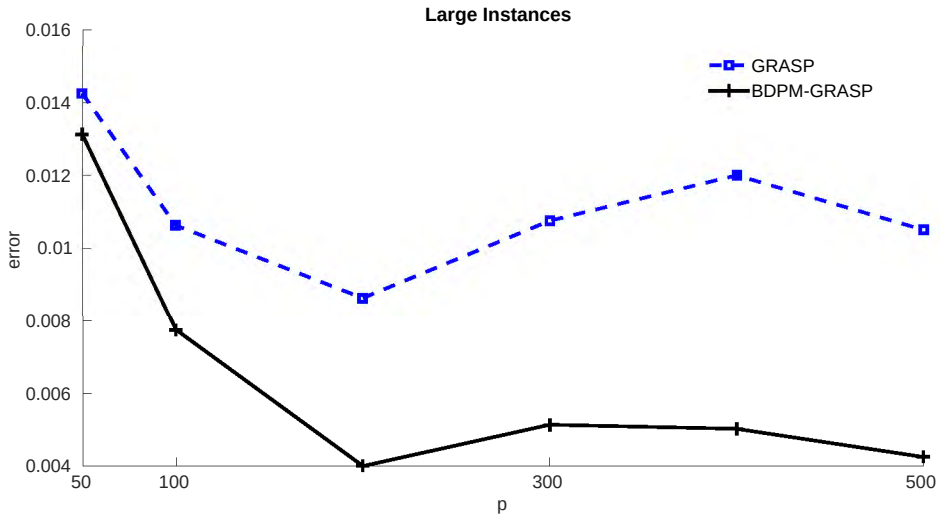


Figure 3.3 : Comparison of BDPM-GRASP to GRASP for large TSPLIB instances. This plot compares average relative error for BDPM-GRASP (solid line, +) the best GRASP run (dashed line, □) for different values of p and test instances from the TSPLIB with at least 1000 vertices.

Table 3.5 : Results for OR-Library instances using branch decompositions of four GRASP heuristic runs.

Instance Data			IP Results		BDPM-GRASP			GRASP		imp-GA		HHP	
Name	size	p	cost	time	BW	time	error	time	error	time	error	time	error
pmed26	600	5	9917	1334	5	0.55	1.000	**0.04**	**1.000**	110	1.000	1.32	1.000
pmed27	600	10	8307	672	4	0.14	1.000	**0.03**	**1.000**	194	1.000	1.03	1.000
pmed28	600	60	4498	8.8	14	K	N/A	0.01	1.0004	692	1.0009	**0.48**	**1.000**
pmed29	600	120	3033	6.9	4	1.77	1.000	0.01	1.002	912	1.0046	**0.48**	**1.000**
pmed30	600	200	1989	7.1	3	**0.25**	**1.000**	0.01	1.009	921	1.0065	0.57	1.000
pmed31	700	5	10086	1687	4	0.62	1.000	**0.05**	**1.000**	128	1.000	1.86	1.000
pmed32	700	10	9297	1678	6	0.36	1.000	**0.04**	**1.000**	268	1.000	1.32	1.000
pmed33	700	70	4700	13.1	11	K	N/A	0.01	1.001	1085	1.000	**0.61**	**1.000**
pmed34	700	140	3013	12.1	8	K	N/A	0.01	1.006	1522	1.0033	**0.77**	**1.000**
pmed35	800	5	K	K	3	0.06	1.000	**0.05**	**1.000**	168	1.000	2.45	1.000
pmed36	800	10	K	K	15	K	N/A	**0.05**	**1.000**	350	1.000	1.88	1.000
pmed37	800	80	5057	18.8	16	K	N/A	**0.01**	**1.000**	1652	1.0024	0.80	1.000
pmed38	900	5	11060	6520	4	0.07	1.000	**0.08**	**1.000**	290	1.000	3.80	1.000
pmed39	900	10	K	K	6	0.13	1.000	**0.05**	**1.000**	381	1.000	2.11	1.000
pmed40	900	90	5128	26.1	22	K	N/A	0.02	1.002	2274	1.0027	**1.00**	**1.000**

Note: Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required for the GRASP runs. K indicates that the algorithm ran out of memory. BW stands for the width of the branch decomposition that was used. The column labeled GRASP gives the result of the best of the GRASP runs. Optimal solutions specified in the OR-Library data set were used to calculate error ratios.

Table 3.6 : Results for small TSPLIB instances using branch decompositions of four GRASP heuristic runs.

Instance Data		IP Results		BDPM-GRASP			GRASP		imp-GA		HHP	
Name	p	cost	time	BW	time	error	time	error	time	error	time	error
lin318	5	179778	6.08	3	0.18	1.173	0.01	1.173	**38**	**1.00002**	0.68	1.173
lin318	10	109398	2.73	1	0.01	1.047	0.01	1.047	**58**	**1.00005**	0.25	1.047
lin318	50	40350	3.42	5	0.54	1.010	0.01	1.014	**226**	**1.001**	0.20	1.010
lin318	100	18959	1.61	3	0.05	1.008	0.01	1.010	**186**	**1.0024**	0.13	1.008
rd400	5	68071	13.3	2	0.04	1.118	0.05	1.118	**70**	**1.0004**	1.35	1.118
rd400	10	46082	9.35	6	0.21	1.048	0.02	1.048	**133**	**1.0031**	0.77	1.048
rd400	50	17295	2.87	4	0.34	1.010	0.01	1.011	**273**	**1.0014**	0.30	1.009
rd400	100	10108	2.74	2	0.04	1.007	0.01	1.007	**336**	**1.002**	0.2	1.007
rd400	200	4532	1.93	2	0.08	1.007	0.01	1.012	**293**	**1.0040**	0.33	1.0044
si535	5	81303	11.6	1	0.02	1.042	0.04	1.042	**108**	**1.000**	1.57	1.040
si535	10	69532	13.4	2	0.04	1.022	0.02	1.022	**211**	**1.000**	0.87	1.022
si535	50	47195	4.29	5	0.33	1.002	0.01	1.004	469	1.0019	**0.48**	**1.0017**
si535	100	38281	3.49	4	0.18	1.0003	0.01	1.0009	691	1.0021	**0.39**	**1.000**
si535	200	26992	3.50	3	**0.04**	**1.000**	0.01	1.0005	619	1.0002	0.43	1.000
ali535	5	965580	16.2	1	0.01	1.132	0.07	1.132	**98**	**1.000**	2.41	1.132
ali535	10	629984	15.8	6	3.40	1.080	0.03	1.080	**237**	**1.0006**	1.96	1.079
ali535	50	232100	7.60	6	9.28	1.014	0.01	1.017	**481**	**1.0031**	0.58	1.014
ali535	100	130639	5.92	3	0.32	1.007	0.01	1.008	**759**	**1.0047**	0.53	1.007
ali535	200	53127	5.53	3	0.07	1.0025	0.01	1.005	687	1.0052	**0.53**	**1.0015**
rat575	5	34971	2201	5	0.72	1.116	0.05	1.116	**178**	**1.0032**	2.54	1.116
rat575	10	23637	36.1	9	171	1.051	0.04	1.052	**324**	**1.0012**	2.02	1.051
rat575	50	9860	685	13	K	N/A	0.01	1.021	659	1.0131	**0.73**	**1.010**
rat575	100	6351	14.0	7	413	1.008	0.01	1.015	801	1.0113	**0.57**	**1.006**
rat575	200	3690	6.73	2	0.05	1.003	0.01	1.011	904	1.0079	**0.60**	**1.002**
gr666	5	1491343	36.8	1	0.02	1.128	0.09	1.128	**202**	**1.000**	3.21	1.128
gr666	10	993899	28.8	5	0.36	1.065	0.09	1.065	**328**	**1.0096**	3.14	1.065
gr666	50	401015	14.0	6	31.4	1.007	0.01	1.007	**795**	**1.0039**	0.83	1.006
gr666	100	250318	23.3	7	80.7	1.0023	0.01	1.005	1178	1.0053	**1.01**	**1.0018**
gr666	200	132376	10.4	3	0.09	1.0002	0.01	1.002	1290	1.0079	**0.91**	**1.00002**
gr666	300	75917	5.25	2	0.22	1.0007	0.01	1.005	1036	1.0116	**0.95**	**1.0002**
u724	5	268898	127	5	4.53	1.148	0.12	1.148	**243**	**1.0001**	4.97	1.148
u724	10	181564	6328	8	18.3	1.050	0.05	1.050	**457**	**1.0006**	3.80	1.050
u724	50	70291	51.9	9	57.8	1.011	0.01	1.014	**1109**	**1.0031**	1.17	1.010
u724	100	43949	12.9	4	0.74	1.0035	0.01	1.005	**1486**	**1.0028**	0.94	1.0035
u724	200	25756	9.33	3	0.22	1.0002	0.01	1.005	1580	1.0126	**0.77**	**1.000**
u724	300	16909	21.1	3	**0.23**	**1.000**	0.01	1.012	1303	1.0174	1.39	1.0004

Note: Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required for the GRASP runs. K indicates that the algorithm ran out of memory. BW stands for the width of the branch decomposition that was used. The column labeled GRASP gives the result of the best of the GRASP runs. Where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios.

Table 3.7 : Results for large TSPLIB instances using branch decompositions of four GRASP heuristic runs.

Instance Data		IP Results		BDPM-GRASP			GRASP		imp-GA		HHP	
Name	p	cost	time	BW	time	error	time	error	time	error	time	error
dsj1000	5	142469699	157	1	0.06	1.134	0.23	1.134	**402**	**1.000002**	7.00	1.134
dsj1000	10	81510884	87.3	2	0.11	1.091	0.09	1.091	**731**	**1.00005**	4.49	1.091
dsj1000	50	34012410	71.7	5	8.40	1.011	0.02	1.012	**2253**	**1.0073**	2.28	1.010
dsj1000	100	22561557	65.5	6	2.53	1.0060	0.02	1.009	3424	1.0076	**2.06**	**1.0056**
dsj1000	200	13689404	28.7	3	0.53	1.0032	0.02	1.005	3417	1.0140	**1.75**	**1.0031**
dsj1000	300	9431896	30.6	4	0.56	1.004	0.03	1.006	3033	1.0347	**2.06**	**1.002**
dsj1000	400	6694084	10.5	2	0.46	1.003	0.03	1.007	2653	1.0431	**2.04**	**1.002**
dsj1000	500	4706364	17.2	2	0.39	1.003	0.03	1.008	2272	1.0490	**2.54**	**1.002**
pr1002	5	1923290	160	5	11.4	1.198	0.25	1.198	**430**	**1.00002**	9.47	1.198
pr1002	10	1263290	129	1	0.06	1.045	0.10	1.045	**810**	**1.00002**	3.08	1.045
pr1002	50	503512	85.7	6	12.4	1.012	0.02	1.012	**2187**	**1.0040**	2.25	1.010
pr1002	100	331435	33.3	6	7.38	1.003	0.02	1.008	3256	1.0060	**2.04**	**1.001**
pr1002	200	200172	43.4	4	0.49	1.001	0.02	1.006	3435	1.0179	**1.98**	**1.0002**
pr1002	300	139232	17.0	3	**0.55**	**1.0001**	0.03	1.007	3053	1.0388	1.85	1.0001
pr1002	400	104068	36.8	4	0.57	1.001	0.03	1.009	2667	1.0364	**2.00**	**1.0002**
pr1002	500	78383	18.5	2	0.52	1.001	0.03	1.005	2287	1.0252	**2.66**	**1.0001**
rl1304	5	3099632	633	7	42.0	1.137	0.57	1.137	**1066**	**1.00001**	22.73	1.137
rl1304	10	K	K	10	K	N/A	0.27	1.063	**1713**	**1.0012**	14.73	1.063
rl1304	50	795468	161	8	K	N/A	0.04	1.019	**4179**	**1.0058**	3.30	1.013
rl1304	100	491929	448	6	17.3	1.008	0.04	1.013	6048	1.0101	**4.05**	**1.007**
rl1304	200	268735	75.2	5	1.48	1.0044	0.04	1.008	6186	1.0288	**2.94**	**1.0037**
rl1304	300	177445	76.0	5	1.03	1.0033	0.04	1.009	5672	1.0486	**3.17**	**1.0028**
rl1304	400	128418	49.7	3	1.22	1.0017	0.05	1.012	5139	1.0811	**3.86**	**1.0016**
rl1304	500	97084	73.2	3	2.48	1.002	0.04	1.013	4675	1.0790	**4.44**	**1.001**
nrw1379	5	433349	384	2	0.14	1.124	0.50	1.124	**1106**	**1.0003**	17.08	1.124
nrw1379	10	K	K	9	29.2	1.052	0.22	1.052	**2021**	**1.0006**	15.43	1.052
nrw1379	50	K	K	15	K	N/A	0.07	1.015	**5406**	**1.0098**	7.36	1.012
nrw1379	100	K	K	14	K	N/A	0.04	1.016	7597	1.0079	**3.78**	**1.005**
nrw1379	200	K	K	8	K	N/A	0.04	1.011	7006	1.0251	**4.62**	**1.003**
nrw1379	300	K	K	4	2.84	1.003	0.04	1.009	6456	1.0393	**4.42**	**1.001**
nrw1379	400	K	K	3	1.36	1.002	0.05	1.006	5870	1.0462	**4.28**	**1.001**
nrw1379	500	K	K	4	0.53	1.002	0.06	1.006	5352	1.0481	**4.30**	**1.001**
fl1400	5	174844	311	2	0.13	1.468	0.26	1.468	**680**	**1.0040**	9.83	1.468
fl1400	10	100872	210	2	0.82	1.082	0.16	1.082	**1504**	**1.0035**	6.93	1.082
fl1400	50	28837	121	6	22.6	1.009	0.05	1.013	4320	1.0111	**3.55**	**1.009**
fl1400	100	K	K	8	132	1.0009	0.04	1.005	6577	1.0124	**4.96**	**1.0005**
fl1400	200	K	K	9	K	N/A	0.04	1.006	7243	1.0139	**4.10**	**1.002**
fl1400	300	K	K	10	K	N/A	0.04	1.011	6689	1.0371	**5.66**	**1.005**
fl1400	400	K	K	9	K	N/A	0.05	1.020	6086	1.0652	**5.96**	**1.001**
fl1400	500	K	K	9	K	N/A	0.05	1.025	5557	1.0671	**6.54**	**1.014**

Note: Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required for the GRASP runs. K indicates that the algorithm ran out of memory. BW stands for the width of the branch decomposition that was used. The column labeled GRASP gives the result of the best of the GRASP runs. Where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios.

Table 3.8 : More results for large TSPLIB instances using branch decompositions of four GRASP heuristic runs.

Instance Data		IP Results		BDPM-GRASP			GRASP		imp-GA		HHP	
Name	p	cost	time	BW	time	error	time	error	time	error	time	error
u1432	5	1210479	566	1	0.14	1.113	0.50	1.113	**1063**	**1.0001**	14.13	1.113
u1432	10	850078	600	7	28.4	1.054	0.22	1.054	**2197**	**1.0001**	16.58	1.054
u1432	50	K	K	11	K	N/A	0.04	1.015	**4809**	**1.0075**	4.91	1.011
u1432	100	K	K	9	K	N/A	0.04	1.013	7454	1.0117	**4.55**	**1.006**
u1432	200	K	K	8	K	N/A	0.04	1.016	7511	1.0239	**4.21**	**1.003**
u1432	300	K	K	12	K	N/A	0.04	1.026	6962	1.0401	**5.79**	**1.004**
u1432	400	K	K	11	K	N/A	0.04	1.028	6414	1.0400	**6.36**	**1.007**
u1432	500	K	K	9	K	N/A	0.04	1.001	5879	1.0040	**4.20**	**1.000**
vm1748	5	K	K	1	0.21	1.155	0.64	1.155	**1657**	**1.00002**	32.26	1.155
vm1748	10	K	K	7	20.0	1.057	0.34	1.057	**3083**	**1.0018**	26.2	1.057
vm1748	50	K	K	9	K	N/A	0.09	1.014	**7708**	**1.0055**	8.40	1.014
vm1748	100	K	K	7	825	1.007	0.05	1.009	12091	1.0067	**6.00**	**1.006**
vm1748	200	K	K	5	4.63	1.0020	0.06	1.007	11628	1.0254	**5.50**	**1.0017**
vm1748	300	K	K	6	7.05	1.0010	0.06	1.007	10933	1.0511	**7.22**	**1.0006**
vm1748	400	K	K	5	4.92	1.0002	0.07	1.006	10140	1.0727	**6.88**	**1.0001**
vm1748	500	K	K	5	3.00	1.001	0.08	1.008	9477	1.0803	**7.58**	**1.0003**
d2103	5	K	K	1	0.3	1.122	1.20	1.122	**2874**	**1.0006**	37.27	1.122
d2103	10	K	K	9	149	1.048	0.41	1.048	**3767**	**1.0012**	25.6	1.048
d2103	50	K	K	16	K	N/A	0.11	1.014	11033	1.0117	**13.7**	**1.009**
d2103	100	K	K	21	K	N/A	0.07	1.012	17721	1.0095	**9.51**	**1.003**
d2103	200	K	K	8	K	N/A	0.06	1.010	16859	1.0241	**10.65**	**1.001**
d2103	300	K	K	7	62.7	1.003	0.06	1.011	16149	1.0356	**11.87**	**1.001**
d2103	400	K	K	8	24.0	1.004	0.07	1.008	15344	1.0382	**11.99**	**1.002**
d2103	500	K	K	5	8.69	1.007	0.08	1.018	14450	1.0459	**15.61**	**1.005**

Note: Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required for the GRASP runs. K indicates that the algorithm ran out of memory. BW stands for the width of the branch decomposition that was used. The column labeled GRASP gives the result of the best of the GRASP runs. Where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios.

Table 3.9 : Results for instances using branch decompositions of less than four GRASP heuristic runs.

Instance Data			BDPM-GRASP					GRASP		imp-GA		HHP	
Name	size	p	runs	BW	cost	time	error	time	error	time	error	time	error
pmed28	600	60	2	6	4500	3.73	1.0004	0.01	1.0004	692	1.0009	**0.48**	**1.000**
pmed33	700	70	2	5	4700	14.9	1.000	0.01	1.001	1085	1.000	**0.61**	**1.000**
pmed34	700	140	3	4	3014	2.74	1.0003	0.01	1.006	1522	1.0033	**0.77**	**1.000**
pmed36	800	10	2	10	9934	564	1.000	**0.05**	**1.000**	350	1.000	1.88	1.000
pmed37	800	80	2	16	K	K	N/A	0.01	1.000	1652	1.0024	0.80	1.000
pmed40	900	90	2	8	K	K	N/A	0.02	1.002	2274	1.0027	**1.00**	**1.000**
rat575	575	50	2	5	10064	4.18	1.021	0.01	1.021	659	1.0131	**0.73**	**1.010**
rl1304	1304	10	3	7	2266366	16.9	1.063	0.27	1.063	**1713**	**1.0012**	14.73	1.063
rl1304	1304	50	3	6	807157	15.9	1.015	0.04	1.019	**4179**	**1.0058**	3.30	1.013
nrw1379	1379	50	2	4	132525	5.50	1.016	0.06	1.016	**5406**	**1.0098**	7.36	1.012
nrw1379	1379	100	2	5	89757	26.6	1.016	0.03	1.017	7597	1.0079	**3.78**	**1.005**
nrw1379	1379	200	3	6	58789	287	1.006	0.04	1.011	7006	1.0251	**4.62**	**1.003**
fl1400	1400	200	3	7	9205	212	1.003	0.04	1.006	7243	1.0139	**4.10**	**1.002**
fl1400	1400	300	3	7	6520	427	1.008	0.04	1.011	6689	1.0371	**5.66**	**1.005**
fl1400	1400	400	3	7	4953	172	1.015	0.04	1.020	6086	1.0652	**5.96**	**1.001**
fl1400	1400	500	3	7	3934	110	1.018	0.04	1.025	5557	1.0671	**6.54**	**1.014**
u1432	1432	50	2	5	367198	3.40	1.014	0.04	1.015	**4809**	**1.0075**	4.91	1.011
u1432	1432	100	2	4	246988	9.32	1.012	0.04	1.013	7454	1.0117	**4.55**	**1.006**
u1432	1432	200	3	7	161386	201	1.009	0.04	1.016	7511	1.0239	**4.21**	**1.003**
u1432	1432	300	2	3	125997	3.41	1.019	0.04	1.028	6962	1.0401	**5.79**	**1.004**
u1432	1432	400	3	7	104758	144	1.013	0.04	1.028	6414	1.0400	**6.36**	**1.007**
u1432	1432	500	2	3	93200	**2.66**	**1.000**	0.04	1.002	5879	1.0040	4.20	1.000
vm1748	1748	50	3	7	1018934	683	1.014	0.09	1.014	**7708**	**1.0055**	8.40	1.014
d2103	2103	50	2	5	306145	27.4	1.014	0.11	1.014	11033	1.0117	**13.7**	**1.009**
d2103	2103	100	2	5	196578	68.8	1.009	0.07	1.013	17721	1.0095	**9.51**	**1.003**
d2103	2103	200	3	8	118888	23.3	1.004	0.06	1.010	16859	1.0241	**10.65**	**1.001**
pcb3038	3038	10	3	7	1297532	216	1.0687	1.27	1.0687	**9929**	**1.000**	90.50	1.0680
pcb3038	3038	50	2	7	K	K	N/A	0.24	1.0093	**27316**	**1.000**	26.67	1.003
pcb3038	3038	100	2	5	357653	138	1.0091	0.15	1.0098	37658	1.006	**21.69**	**1.000**
pcb3038	3038	200	2	5	240436	173	1.006	0.13	1.007	36296	1.017	**19.02**	**1.000**
pcb3038	3038	300	2	4	189091	58.3	1.006	0.15	1.009	35052	1.043	**16.43**	**1.000**
pcb3038	3038	400	2	4	157828	13.6	1.004	0.16	1.009	33699	1.067	**17.94**	**1.000**
pcb3038	3038	500	3	7	135969	57	1.003	0.19	1.009	32542	1.084	**17.79**	**1.000**

Note: Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required for the GRASP runs. K indicates that the algorithm ran out of memory. BW stands for the width of the branch decomposition that was used. The column labeled GRASP gives the result of the best of the GRASP runs. For TSPLIB instances where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios. For the pcb3038 instances the PMPLP could not be solved. So, I use the best available solution to calculate the error ratios.

3.5.2 BDPM-LP

For BDPM-LP, I use the linear programming support graph to form the support graph for my algorithm. A variable was considered to be positive and its corresponding edge was in the support graph if the variable had a value at least 0.0001 in the linear programming solution. As with BDPM-GRASP, there were instances for which the width of the decomposition of the support graph was too high for BDPM-LP to be practical. Also, for some instances a feasible p-median solution did not exist on the linear programming support graph. In either case, I used lexicographic fixing, as explained in Section 3.3.1, and resolved the linear program until a suitable support graph was obtained. I compare the results of BDPM-LP to integer programming, imp-GA, and to the HHP algorithm.

The results of BDPM-LP show that my algorithm obtains high quality solutions and usually gets a better quality solution than both HHP and imp-GA. Table 3.10 gives average results for the different methods. The BDPM-LP algorithm does have a much higher computational cost than HHP since the PMPLP must be solved. However, when the linear program solution is already known, and the width of the branch decomposition of the support graph is at most 5, then BDPM-LP is faster than the HHP algorithm. Over all the TSPLIB instances, the average time required by the BDPM portion of BDPM-LP was 106 seconds compared to 4.8 seconds for the HHP algorithm. However, for the instances that did not require fixing, the average time required was 10.2 seconds compared to 3.6 seconds for the HHP algorithm. If we only consider instances that did not require fixing and had width at most 5, then the average time required by the BDPM portion of BDPM-LP was 1.2 seconds compared to 3.7 seconds for the HHP algorithm. So, if the root relaxation solution is already known and it has a decomposition of width at most 5,

then BDPM-LP is actually faster than the HHP algorithm. The BDPM-LP method also outperforms the imp-GA method. In contrast to the results for BDPM-GRASP, Table 3.10 shows that BDPM-LP has less error than imp-GA even when results are averaged over all p values.

For most of the OR-Library instances, the HHP algorithm acheives an optimum solution, and the PMPLP solution is either integral or has a high width decomposition. Table 3.13 lists the widths for these instances. Because they either could not be solved or were solved by fixing without using the branch decomposition algorithm, I do not report results for these instances.

For the small (less than 1000 vertices) TSPLIB instances where the linear program solution was not integral, the average error ratio of BDPM-LP was 1.000783 compared to 1.0147 for the HHP algorithm and 1.0063 for imp-GA. On average for small instances, BDPM-LP had 79.2% less error than the HHP algorithm and 64.9% less error than imp-GA. Detailed results for these instances are reported in Table 3.14. Thus, BDPM-LP outperforms both HHP and imp-GA on these instances.

For the large (at least 1000 vertices) TSPLIB instances where the linear program solution was not integral, the average error ratio of BDPM-LP was 1.00238 compared to 1.00951 for the HHP algorithm and 1.0266 for imp-GA. On average for large instances, BDPM-LP had 33.3% less error than the HHP algorithm and 86.0% less error than imp-GA. Detailed results for these instances are reported in Table 3.15. Thus, BDPM-LP outperforms both HHP and imp-GA on these instances.

Much of the additional error for the large instances appears to be caused by fixing when the width of the linear program solution support graph decomposition was too high. If we remove the instances where I used fixing to reduce the width,

then BDPM-LP had an average error ratio of 1.000413 compared to an average error ratio of 1.00656 for the HHP algorithm and 1.0290 for imp-GA. On average for these instances, BDPM-LP had 44.6% less error than the HHP algorithm and 96.4% less error than imp-GA.

As with the BDPM-GRASP results, I break the average results down by p values. Tables 3.11 and 3.12 and Figures 3.4 and 3.5 show average relative errors for different p values. Unlike BDPM-GRASP, BDPM-LP performs well for all p-values. For p values of 400 and 500, the linear program support graphs often had high width that was corrected with fixing. This fixing leads to higher errors for BDPM-LP for these p-values. However, Figure 3.6 shows that BDPM-LP still outperforms HHP for instances that did not require fixing to correct high widths. BDPM-LP also outperforms imp-GA across almost all p-values, with imp-GA only having less error for the small instances with p values either 5 or 10.

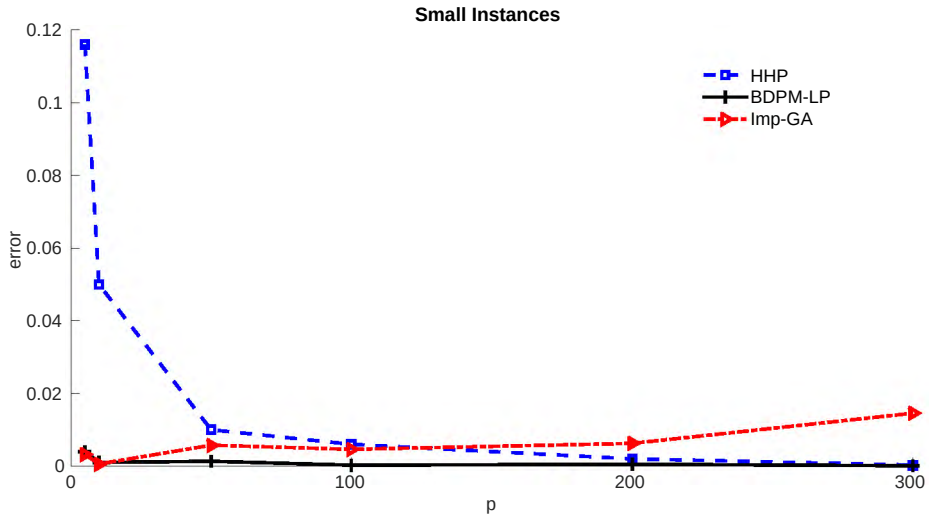


Figure 3.4 : Comparison of BDPM-LP to HHP and imp-GA for small TSPLIB instances. This plot compares average relative error for BDPM-LP (solid line, +), HHP (dashed line, □), and imp-GA (dash-dotted line, ▶) for different values of p and test instances from the TSPLIB with less than 1000 vertices.

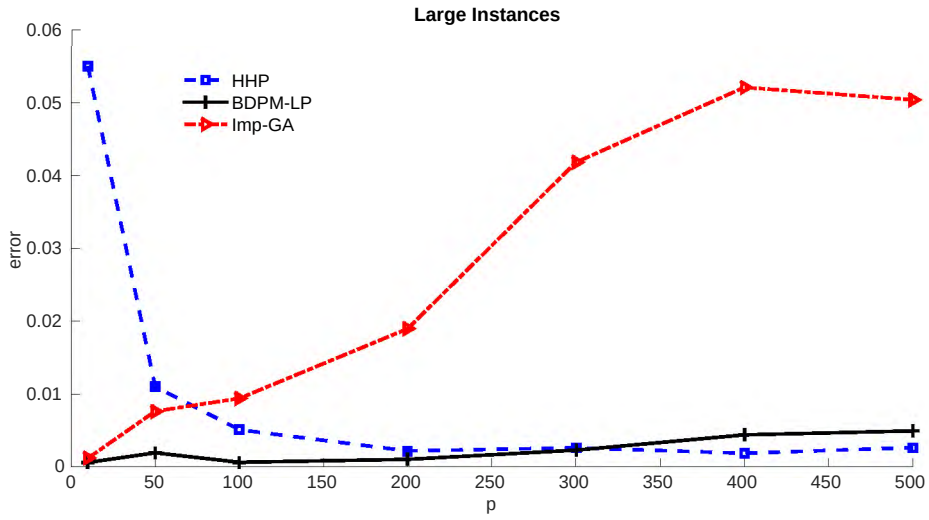


Figure 3.5 : Comparison of BDPM-LP to HHP and imp-GA for large TSPLIB instances. This plot compares average relative error for BDPM-LP (solid line, +), HHP (dashed line, □), and imp-GA (dash-dotted line, ▶) for different values of p and test instances from the TSPLIB with at least than 1000 vertices.

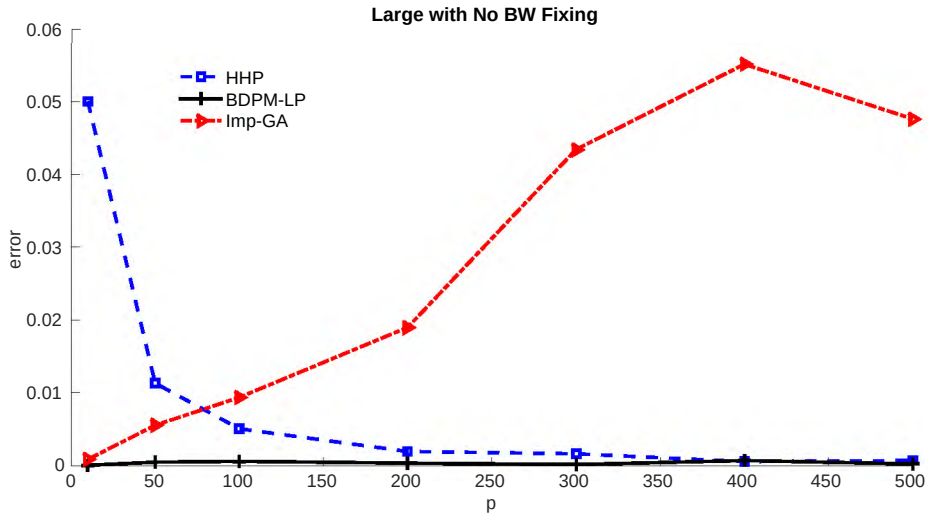


Figure 3.6 : Comparison of BDPM-LP to HHP and imp-GA for large TSPLIB instances that did not require fixing to correct high width. This plot compares average relative error for BDPM-LP (solid line, +), HHP (dashed line, □), and imp-GA (dash-dotted line, ►) for different values of p and test instances from the TSPLIB with at least than 1000 vertices that did not require fixing to correct high widths.

Table 3.10 : Average results for BDPM-LP.

Algorithm	Small TSP (16 instances)			Large TSP (39 instances)		
	# solved	time	error	# solved	time	error
BDPM-LP	16	25.2	1.0008	39	493	1.0024
HHP	16	0.97	1.0147	39	6.38	1.0095
imp-GA	16	713	1.0063	39	5583	1.0266
IP	16	585	1.0	12	83	1.0

Note: The # solved column indicates the number of instances that the relevant algorithm solved without exhausting available memory. Time is reported in seconds.

Table 3.11 : Average relative errors for BDPM-LP on small TSPLIB instances broken down by p.

Algorithm	p=5	p=10	p=50	p=100	p=200	p=300
BDPM-LP	0.0040	0.0010	**0.0013**	**0.0003**	**0.0005**	**0.0001**
HHP	0.1160	0.0500	0.0100	0.0060	0.0020	0.0003
imp-GA	**0.0032**	**0.0006**	0.0058	0.0046	0.0062	0.0145

Note: BDPM-LP performs better than HHP for all p values, and better than imp-GA for p at least 50. Bold text indicates the method that had the least error.

Table 3.12 : Average relative errors for BDPM-LP on large TSPLIB instances broken down by p.

Algorithm	p=10	p=50	p=100	p=200	p=300	p=400	p=500
BDPM-LP	**0.0006**	**0.0019**	**0.0006**	**0.0010**	**0.0023**	0.0044	0.0049
HHP	0.0550	0.0110	0.0051	0.0022	0.0026	**0.0019**	**0.0026**
imp-GA	0.0012	0.0076	0.0094	0.0190	0.0418	0.0521	0.0504

Note: BDPM-LP performs better than HHP for p less than 400, and better than imp-GA for all p values. Bold text indicates the method that had the least error.

Table 3.13 : Branch-Widths of PMPLP Support Graphs for OR-Library Instances.

Instance Data			IP Results		
Name	size	p	cost	time	Width
pmed26	600	5	9917	1334.1	22
pmed27	600	10	8307	671.8	33
pmed28	600	60	4498	8.8	1
pmed29	600	120	3033	6.9	1
pmed30	600	200	1989	7.1	1
pmed31	700	5	10086	1686.8	7
pmed32	700	10	9297	1677.6	34
pmed33	700	70	4700	13.1	1
pmed34	700	140	3013	12.1	1
pmed35	800	5	K	K	10
pmed36	800	10	K	K	40
pmed37	800	80	5057	18.8	2
pmed38	900	5	11060	6520	23
pmed39	900	10	K	K	34
pmed40	900	90	5128	26.1	1

Note: Most of these instances are either integral (indicated by a width of 1) or have width much higher than 7.

Table 3.14 : Results for small TSPLIB instances using branch decompositions of the PMPLP support graph.

Instance Data		IP Results		BDPM-LP				HHP Algorithm		imp-GA	
Name	p	cost	time	width	LP time	BD time	error	time	error	time	error
lin318	50	40350	3.42	2^F	1.02	0.53	1.002	0.20	1.010	**226**	**1.001**
lin318	100	18959	1.61	2	**0.76**	**0.04**	**1.000**	0.13	1.008	186	1.0024
rd400	100	10108	2.74	2	**1.39**	**0.07**	**1.000**	0.20	1.007	336	1.002
rd400	200	4532	1.93	2	**0.91**	**0.05**	**1.0007**	0.33	1.004	293	1.004
ali535	100	130639	5.92	2	**3.41**	**0.11**	**1.000**	0.53	1.007	759	1.0047
ali535	200	53127	5.53	3	**2.31**	**0.16**	**1.001**	0.53	1.002	687	1.0052
rat575	5	34971	2201	2^{BW}	62.5	12.6	1.004	2.54	1.116	**178**	**1.0032**
rat575	50	9860	685	7^{BW}	**17.3**	**93.3**	**1.002**	0.73	1.010	659	1.0131
rat575	100	6351	14.0	3^F	**9.83**	**1.85**	**1.0003**	0.57	1.006	801	1.0113
rat575	200	3690	6.73	2	**6.50**	**0.16**	**1.0003**	0.60	1.002	904	1.0079
gr666	100	250318	23.3	2^F	**8.72**	**0.74**	**1.001**	1.01	1.002	1178	1.0053
gr666	200	132376	10.4	2^F	**4.98**	**1.32**	**1.00002**	0.91	1.00002	1290	1.0079
gr666	300	75917	5.25	2	**2.75**	**0.10**	**1.000**	0.95	1.0002	1036	1.0116
u724	10	181564	6328	1^F	93.6	14.8	1.001	3.80	1.050	**457**	**1.0006**
u724	50	70291	51.9	6	**24.5**	**27.6**	**1.00001**	1.17	1.010	1109	1.0031
u724	300	16909	21.1	2	**9.48**	**0.15**	**1.0002**	1.39	1.0004	1303	1.0174

Note: Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The LP time column gives the time required to solve the root relaxation. The BD time column gives the time required for BDPM-LP after the root relaxation was solved. K indicates that the algorithm ran out of memory. If fixing was required, I indicate that with superscripts on the width. The BW superscript indicates that I fixed because of high widths, and the F superscript indicates that I fixed because the PMPLP support graph did not contain a feasible solution. Where the PMPIP could not be solved, the PMPLP solution cost was used to calculate the error ratios. Instances for which the PMPLP solution was integral are not reported.

Table 3.15 : Results for large TSPLIB instances using branch decompositions of the PMPLP support graph.

Instance Data		IP Results		BDPM-LP				HHP Algorithm		imp-GA	
Name	p	cost	time	width	LP time	BD time	error	time	error	time	error
dsj1000	50	34012410	71.7	2^F	**53.2**	**3.36**	**1.001**	2.28	1.010	2253	1.0073
dsj1000	100	22561557	65.5	5	**39.1**	**0.83**	**1.0003**	2.06	1.006	3424	1.0076
dsj1000	200	13689404	28.7	2	**23.3**	**0.52**	**1.000**	1.75	1.003	3417	1.0140
dsj1000	300	9431896	30.6	2^F	**15.1**	**5.26**	**1.0002**	2.06	1.002	3033	1.0347
dsj1000	500	4706364	17.2	2	**6.4**	**0.26**	**1.0003**	2.54	1.002	2272	1.0490
pr1002	50	503512	85.7	2^F	**70.0**	**1.64**	**1.0003**	2.25	1.010	2187	1.0040
pr1002	200	200172	43.4	2^F	**18.9**	**3.85**	**1.0001**	1.98	1.0002	3435	1.0179
pr1002	400	104068	36.8	2^F	25.4	3.22	1.0003	**2.00**	**1.0002**	2667	1.0364
pr1002	500	78383	18.5	2^F	4.99	2.04	1.0005	**2.66**	**1.0001**	2287	1.0252
rl1304	10	K	K	1^{FBW}	1305	119	1.002	14.73	1.063	**1713**	**1.0012**
rl1304	100	491929	448	3	**109**	**0.75**	**1.0001**	4.05	1.007	6048	1.0101
rl1304	300	177445	76.0	2	**58.1**	**1.2**	**1.000**	3.17	1.003	5672	1.0486
rl1304	500	97084	73.2	2	**46.4**	**1.38**	**1.0001**	4.44	1.001	4675	1.0790
nrw1379	10	K	K	3	**714**	**0.73**	**1.00005**	15.43	1.052	2021	1.0006
nrw1379	50	K	K	1^{BW}	**437**	**460**	**1.004**	7.36	1.012	5406	1.0098
nrw1379	100	K	K	6	**109**	**157**	**1.001**	3.78	1.005	7597	1.0079
nrw1379	200	K	K	7	**183**	**45.1**	**1.001**	4.62	1.0027	7006	1.0251
nrw1379	300	K	K	2	**103**	**1.26**	**1.0003**	4.42	1.001	6456	1.0393
nrw1379	400	K	K	2^F	**67.3**	**19.1**	**1.0005**	4.28	1.001	5870	1.0462
nrw1379	500	K	K	2^F	**54.0**	**12.0**	**1.0003**	4.30	1.001	5352	1.0481
fl1400	100	K	K	5	**95.8**	**1.81**	**1.0003**	4.96	1.0005	6577	1.0124
fl1400	200	K	K	6^{BW}	145	169	1.003	**4.10**	**1.002**	7243	1.0139
fl1400	300	K	K	2^{FBW}	120	313	1.012	**5.66**	**1.005**	6689	1.0371
fl1400	400	K	K	5^F	89.4	20.8	1.002	**5.96**	**1.001**	6086	1.0652
fl1400	500	K	K	3^{FBW}	128	509	1.033	**6.54**	**1.014**	5557	1.0671
u1432	50	K	K	1^{FBW}	**273**	**193**	**1.003**	4.91	1.011	4809	1.0075
u1432	100	K	K	4^F	**149**	**38.4**	**1.001**	4.55	1.006	7454	1.0117
u1432	200	K	K	2^{FBW}	**121**	**56.7**	**1.001**	4.21	1.003	7511	1.0239
u1432	300	K	K	7^{BW}	**107**	**9.10**	**1.001**	5.79	1.004	6962	1.0401
u1432	400	K	K	5^{BW}	163	523	1.019	**6.36**	**1.007**	6414	1.0400
u1432	500	K	K	3	62.0	2.14	1.000	**4.20**	**1.000**	5879	1.0040
vm1748	10	K	K	4^{BW}	**1640**	**87.7**	**1.0002**	26.2	1.057	3083	1.0018
vm1748	50	K	K	3	**232**	**1.75**	**1.0002**	8.40	1.014	7708	1.0055
vm1748	100	K	K	5	**268**	**6.86**	**1.001**	6.00	1.006	12091	1.0067
vm1748	300	K	K	4	**118**	**2.52**	**1.0002**	7.22	1.0006	10933	1.0511
vm1748	400	K	K	2^F	80.7	4.56	1.0001	**6.88**	**1.0001**	10140	1.0727
vm1748	500	K	K	2	80.2	2.96	1.0003	**7.58**	**1.0003**	9477	1.0803
d2103	10	K	K	1^F	**2769**	**33.1**	**1.0001**	25.6	1.048	3767	1.0012
d2103	50	K	K	1^{BW}	**3396**	**2863**	**1.003**	13.7	1.009	11033	1.0117

Note: Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The LP time column gives the time required to solve the root relaxation. The BD time column gives the time required for BDPM-LP after the root relaxation was solved. K indicates that the algorithm ran out of memory. If fixing was required, I indicate that with superscripts on the width. The BW superscript indicates that I fixed because of high widths, and the F superscript indicates that I fixed because the PMPLP support graph did not contain a feasible solution. Where the PMPIP could not be solved, the PMPLP solution cost was used to calculate the error ratios. Instances for which the PMPLP solution was integral are not reported.

3.6 Conclusions

In this chapter, I introduced BDPM, a branch decomposition based dynamic programming algorithm for the p-median problem. This algorithm works both for finding a high quality solution in the linear programming support graph and for finding an improved solution out of a pool of heuristic solutions. However, BDPM is sensitive to the width of the decomposition that is used and widths above 7 are not feasible on a typical desktop computer due to memory requirements. However, my computational experiments showed that BDPM can perform better than other state-of-the-art methods when these width restrictions are met.

The computational results in Section 3.5 show that my BDPM algorithm is a useful tool for improving on a pool of heuristic solutions. Using BDPM to improve a pool of heuristic solutions works best when p is sufficiently large. BDPM-GRASP significantly improved on GRASP solutions and also had less error than imp-GA for instances with p values of at least 100, while still being significantly faster than integer programming or my implementation of imp-GA. In terms of relative error, imp-GA was the better heuristic when p values were less than 100. The dependence of BDPM-GRASP on high p values corresponds to the observation that, as p increases, the difference between the GRASP runs used to form the support graph likely increases. Thus, the support graph likely has more edges and contains better solutions as p increases. A possible direction for future study is to investigate whether the use of different heuristics leads to a better pool on which to run BDPM.

The results in Section 3.5 also show that BDPM-LP is able to create high quality solutions when the width of the linear program support graph decomposition is at most 7. While slower than the HHP algorithm, BDPM-LP was more accurate. BDPM-LP was also faster and more accurate than my

implementation of imp-GA. However, BDPM-LP requires a solution of the PMPLP, which may become difficult for larger problems. Higher widths are not feasible on a typical desktop computer due to memory requirements, and although these higher widths can be dealt with through fixing, such fixing led to higher errors in my experiments. Thus, a possible direction for future study is to investigate whether different fixing rules can lower the branchwidth of the support graph without hurting solution quality. Also, Theorem 3.2 gives an error bound on the result of BDPM-LP based on the smallest fractional part of the linear program support graph. This bound is not known to be tight and can likely be improved.

Of the methods tested in this chapter, HHP is the best general method for when the integer program is not practical due to time constraints. Imp-GA is best when p is small. However, if the linear program solution is available and has low branch-width, then the BDPM-LP method is better than HHP and Imp-GA. BDPM-GRASP is better than Imp-GA when p is not small and usually is very close to the solution provided by HHP. When the branch decomposition width is less than 5, then BDPM-GRASP is faster than HHP. Also, combining BDPM-GRASP with HHP could potentially lead to an improved method. However, the solutions provided by multiple HHP runs did not have significant variation. Thus, BDPM cannot be applied directly to HHP solutions to improve them in the way that BDPM can be applied to GRASP.

Chapter 4

Zero-Forcing Literature*

The zero-forcing process and zero-forcing number of a graph were introduced by the American Institute of Mathematics Minimum Rank Special Graphs Work Group (Barioli, Barret, Butler, Ciobă, Cvetković, Fallat, Godsil, Haemers, Hogben, Mikkelson, Narayan, Pryporova, Sciriha, So, Stevanović, van der Holst, Vander Muelen, and Wehe) [3] in 2007 to bound the maximum nullity of certain matrices. Given a symmetric matrix A, define $G(A)$ to be the graph corresponding to the adjacency matrix with ones at every nonzero element of A not on the diagonal of A and zeros elsewhere. The AIM Group [3] showed that the zero-forcing number of the graph of a symmetric matrix is an upper bound on the nullity of that symmetric matrix. Furthermore, since any symmetric matrix with the same pattern of nonzero elements will have the same graph, the zero-forcing number of the graph is an upper bound on the maximum nullity attainable by a matrix with that pattern of nonzero elements. This maximum nullity is called the maximum nullity of the graph. The AIM Group [3] used the zero-forcing number to prove bounds on the maximum nullity of various special classes of graphs, such as Cartesian product graphs, Möbius ladders, and block-clique graphs. They also showed that the zero-forcing number is equal to maximum nullity for certain classes of graphs, such as trees, cliques, cycles, and paths.

In the same year that the AIM Group [3] introduced the zero-forcing process and

*This chapter is expanded from [67].

zero-forcing number, Burgarth and Giovannetti [31] studied the same process under the name of graph infection and showed that it was theoretically useful for the control of quantum systems. In their application, the graph to be studied comes from the quantum coupling of a physical system. The spins of the particles in the quantum system are represented by the vertices and the edges are derived from the Hamiltonian representing the system. The goal of the application is to control the spins on the whole system by enforcing a certain spin configuration on some subset of the particles. Burgarth and Giovannetti [31] showed that if a certain spin configuration is enforced on a zero-forcing set of the graph, then all other spins in the graph will be forced to take the same spin configuration. According to Burgarth and Giovannetti [31], although a large-scale, practical application of their research is beyond the capabilities of experimental physicists at the present time, the zero-forcing process may prove useful for controlling quantum memory in a quantum computer.

Since Burgarth and Giovannetti's [31] introduction, the quantum control application of zero-forcing has received further study. Burgarth and Maruyama [33] showed that the zero-forcing process was also useful for identifying the Hamiltonian governing a quantum system. Burgarth, Maruyama, and Nori [34] extended Burgarth and Maruyama's [33] method to work with more complicated types of Hamiltonians. Burgarth, D'Alessandro, Hogben, Severini, and Young [30] showed that the criterion of quantum controllability was equivalent to a criterion for control of linear dynamical systems. Thus, the zero-forcing process can be used to control other dynamical systems on networks, such as those that arise in social networks and robotics. Liu, Slotine, and Barabási [94] studied a weaker version of control for dynamical systems on directed networks and showed that this weaker control

process can be related to maximum matchings. In constrast, the zero-forcing process implies a stronger version of controllability [30], and Monshizadeh, Zhang, and Camlibel [101] showed that, for directed graphs, the number of vertices required for this stronger version of controllability is equal to the zero-forcing number of the graph. Trefois and Delvenne [130] showed that the zero-forcing number of a graph is equal to the size of a maximum constrained matching on a related bipartite graph.

In order to extend the zero-forcing process to work on quantum systems that cannot be modeled as graphs, Puchala [106] generalized the zero-forcing process to hypergraphs. A hypergraph is like a graph except the edges are sets of one or more vertices instead of being limited to two vertices. In the hypergraph setting, the zero-forcing infection rule introduced by Puchala [106] is that a set of infected vertices infects a set of uninfected vertices if that uninfected set is in the same edge as the infected set and that edge is the only edge that contains vertices from the infected set and also uninfected vertices. Thus, the zero-forcing process is basically the same. Infection still proceeds from infected vertices through an edge that is the only edge joining the infected vertex to an uninfected vertex.

Another application for the zero-forcing process was given by Burgarth, Giovannetti, Hogben, Severini, and Young [32]. These authors showed that zero-forcing could be used to build logic circuits capable of evaluating any boolean function. For example, the simple AND and OR operators can be encoded by the graphs in Figure 4.1. In each of these logic gates, the inputs are given by coloring the X (or Y) vertex black if the statement X (or Y) is true and leaving it white if the statement is false. The output of the gate is given by the coloring of the X AND Y vertex at the end of the zero-forcing process. If the X AND Y vertex is colored black by the process, then the statement X AND Y is true. Otherwise, the

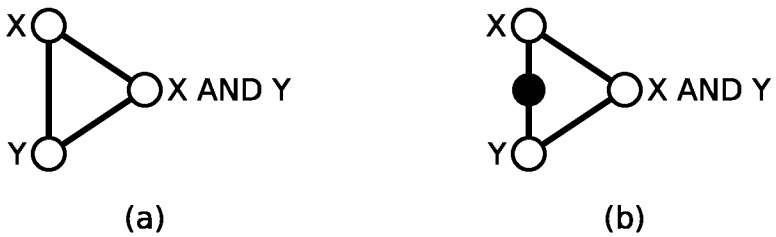


Figure 4.1 : Burgarth et al. [32]. The graph labeled (a) is an AND logic gate. The graph labeled (b) is an OR logic gate.

statement is false. The OR gate works in a similar fashion except there is an extra vertex that is always colored black.

As mentioned previously, Puchala [106] gave a generalization of the zero-forcing process to hypergraphs. Additional generalizations have been introduced in the mathematics literature. Kang and Yi [87] introduced the *probabalistic* zero-forcing number. In probabalistic zero-forcing, each infected vertex, v, infects its neighbors independently with a probability corresponding to the number of neighbors of v that are infected. This probability is 1 if v has only one uninfected neighbor. If two infected vertices have a common neighbor, then they act independently to try to infect that neighbor. The zero-forcing rule can be recovered from probabalistic zero-forcing by changing the probability of infection to be 0 if a vertex has more than one uninfected neighbor.

Amos, Caro, Davila, and Pepper [6] introduced another generalization of zero-forcing called the k-forcing number. In this generalization, an infected vertex with k or less uninfected neighbors infects all of those uninfected neighbors, but if it has more than k uninfected neighbors then it cannot infect any of them. Obviously, this rule is equivalent to zero-forcing if $k = 1$. Amos et al. [6] also gave several

upper bounds on the k-forcing number of a graph based on maximum degree and connectivity.

In addition to the generalizations, some modifications to the zero-forcing rule have led to interesting graph parameters. Barioli, Barrett, Fallat, Hall, Hogben, Shader, van den Driessche, and van der Holst [17] introduced the *positive semidefinite* zero-forcing rule. In this rule, at each iteration of the infection rule, the uninfected vertices are associated to the component of the graph that would contain them if all the infected vertices were removed from the graph. Then, an infected vertex, v, will infect an uninfected vertex, w, if that w is the only uninfected neighbor of v that is in the component containing w. The *positive semidefinite* zero-forcing number is analogous to the zero-forcing number of a graph. It is the smallest set of initially infected vertices that will infect the entire graph through the positive semidefinite zero-forcing process. Barioli et al. [17] showed that the positive semidefinite zero-forcing number provides at least as good an upper bound on the nullity of Hermitian positive semidefinite matrices as the zero-forcing number.

In a different paper, Barioli et al. [18] also introduced infection rules for the *enhanced* zero-forcing number and the *loop* zero-forcing number. They also provided infection rules for minor monotone floors of each type of zero-forcing number (zero-forcing, positive semidefinite, enhanced, loop). These different types of zero-forcing numbers are each relevant to minimum rank for certain special types of graphs, but they are also interesting for their relationship to certain other graph parameters. For example, Barioli et al. [18] showed that the minor monotone floor of the loop zero-forcing number is the pathwidth of the graph. They also gave a treewidth zero-forcing rule for which the treewidth zero-forcing number is equal to the treewidth of the graph.

Of the zero-forcing modifications, the positive semidefinite zero-forcing number has seen the most study. Ekstrand, Erickson, Hay, Hogben, and Roat [59] extended the definition of the positive semidefinite zero-forcing number to multigraphs. Multigraphs may have more than one edge between two vertices. Ekstrand et al.'s [59] extension adds the additional restriction that an infected vertex can only infect a neighbor if it is joined to that neighbor by only one edge. Ekstrand, Erickson, Hall, Hay, Hogben, Johnson, Kingsley, Osborne, Peters, Roat, Ross, Row, Warnberg, and Young [58] characterized graphs that had positive semidefinite zero-forcing number of 2 and $|V| - 2$. Fallat, Meagher, and Yang [65] gave a linear time theoretical algorithm for determining the positive semidefinite zero-forcing number of chordal graphs; however, they did not implement their algorithm.

Although the zero-forcing process has been studied heavily since its introduction, almost all of those studies have focused on special types of graphs. In fact, there has been a proliferation of papers providing bounds on the zero-forcing number for special types of graphs. Table 4.1 provides a listing of these papers and the types of graphs studied.

Several papers have also focused on bounding the zero-forcing number for simple, undirected graphs. However, these bounds have been very weak. Eroh, Kang, and Yi [63] showed that if a graph has a connected complement graph, then the zero-forcing number of the graph is at most $|V| - 3$. Davila and Kenter [50] showed that for graphs with girth (size of smallest cycle) at least 5, the zero-forcing number is at least $2\delta - 2$ where δ is the minimum degree of a vertex in the graph. Amos, Caro, Davila, and Pepper [6] showed that for connected graphs with maximum degree (Δ) at least 2, the zero-forcing number is at most $\frac{(\Delta-2)|V|+2}{\Delta-1}$. Very recently, Gentner, Penso, Rautenbach, and Souza [77] proved a conjecture by Davila

Table 4.1 : Literature on zero-forcing for special types of graphs

Author(s)	Year	Graph Types
Severini [124]	2008	trees
Almodovar, DeLoss, Hogben, et al. [4]	2010	Ciclos and Estrellas
Huang, Chang, Yeh [86]	2010	Block-clique, interval, and product
Row [118]	2011	Cacti
Meyer [97]	2012	Bipartite circulants
Yi [135]	2012	Permutation
Catral, Cepek, Hogben, et al. [39]	2012	Subdivided
Edholm, Hogben, Huynh, et al. [57]	2012	Grids
Eroh, Kang, Yi [62]	2013	Line
Eroh, Kang, Yi [64], [63]	2014	Trees and Unicyclic
Taklimi, Fallat, and Meagher [126]	2014	Block-cycle and outerplanar
Barrett, Butler, Catral, et al. [19]	2014	Complete subdivision
Berliner, Brown, Carlson, et al. [22]	2015	Oriented

and Kenter [50] that the zero-forcing number is at least $2\delta - 2$ for any graph with girth at least 4 and minimum degree at least 2. While these are interesting theoretical bounds, they are not tight enough to be useful in computing zero-forcing numbers for most graphs.

The zero-forcing iteration index has received less attention than the zero-forcing number. Chilakamarri, Dean, Kang, and Yi [40] defined the iteration index and gave some non-trivial bounds for certain special graph classes, such as trees and cartesian product graphs. Hogben, Huynh, Kingsley, Meyer, Walker, and Young [83] characterized graphs with extreme iteration indices. Warnberg [132] studied the iteration index for positive semidefinite zero-forcing for graphs with extreme iteration indices. Butler and Young [36] gave bounds for an index that minimized the combined sum of the forcing set size and the number of iterations required. To our knowledge, these are the only published papers that deal with the zero-forcing iteration index.

Part of the reason for the limited literature on the iteration index is that the iteration index has been shown to be incomparable to many graph invariants. For example, Hogben et al. [83] gave the counterexample in Figure 4.2 to show that iteration index and diameter are not comparable. I give a different counterexample in Figure 4.3 that shows that these invariants are still not comparable even when the graph has any given minimum degree. Given any desired minumum degree, δ, our counterexample is composed of cliques of size δ connected in series as in Figure 4.3. No matter how many cliques are in the counterexample, the clique on the end is a minimum forcing set of the graph. Thus, for a counterexample built from k cliques, the iteration index is $I(G) = \delta(k-1)$, but the diameter is at most k. Figure 4.3 shows the counterexample for $\delta = 4$.

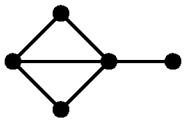


Figure 4.2 : Hogben et al. [83], the dart graph. Note that the diameter is 2, but $I(G) = 3$

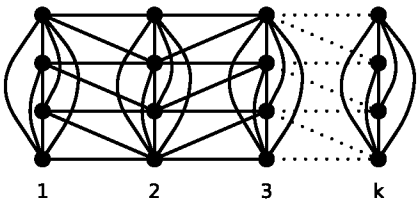


Figure 4.3 : Counterexample for $\delta = 4$. Note that the diameter is at most k, but $I(G) = 4(k-1)$.

There is even less literature on the computation of minimum zero-forcing sets. Computing the zero-forcing number of a simple, undirected graph was shown to be NP-hard by Aazami [1]. Trefois and Delvenne [130] extended this result to directed graphs with loops (a loop is an edge that begins and ends on the same vertex). Row [119] showed that cut vertices (vertices whose removal causes the graph to become disconnected) in a graph could be used to help compute the zero-forcing number. In particular, he related the zero-forcing number of the graph to the zero-forcing numbers of the components left when the cut vertex is removed. However, this result is limited to a single cut vertex, the result has not been shown to hold for cut sets containing more than one vertex.

Aazami [1] also gave a dynamic programming algorithm for computing zero-forcing sets; however, he did not give any computational results for this

algorithm. Furthermore, the complexity of the dynamic programming algorithm is exponential in the square of the treewidth of the graph [1]. Thus, it will be impractical for all but graphs with very small treewidth. Aazami [2] also gave a formulation for the Power Dominating Set problem, which is closely related to the zero-forcing problem. However, he did not report computational results for this formulation.

The only algorithm that has been implemented is the Wavefront Algorithm of Butler, DeLoss, Grout, Hall, LaGrange, McKay, Smith, and Tims [35]. These authors have published little more than the source code for the algorithm. To my knowledge neither a proof of correctness nor an analysis of complexity has been published for this algorithm. I provide both in this thesis.

In addition to modifying the zero-forcing rule, some interesting variants of the zero-forcing problem arise from restricting the zero-forcing set. For example, every vertex in the zero-forcing set may be required to have a neighbor in the zero-forcing set. This restriction is called *total* zero-forcing and was introduced by Davila [46] and since studied by Davila and Henning [47], [49].

Another way to restrict the zero-forcing set is to require that the subgraph induced by the set is connected. This restriction is called *connected* zero-forcing. The connected zero-forcing problem was introduced by Brimkov and Davila [26] who gave formulas for the connected forcing number of trees, snarks, and graphs with a single maximal clique larger than 2 vertices. They also characterized graphs with connected zero-forcing number $|V| - 1$. Davila, Henning, Magnant, and Pepper [48] gave bounds on the connected forcing number based on properties of the graph such as girth, minimum degree, and maximum degree. Brimkov [25] showed that finding a minimum connected zero-forcing set is NP-hard. Brimkov, Fast, and Hicks

[28] also characterized graphs that have extreme connected forcing numbers of 2 or $|V| - 2$. Up to this point, no methods have been developed for computing connected zero-forcing sets for general graphs. One of the contributions of this thesis is to provide such methods. In this thesis, we give integer programming formulations and computational results for both the zero-forcing problem and the connected zero-forcing problem.

Chapter 5

Theory of Minimum Zero-Forcing Sets*

5.1 Introduction

The zero-forcing problem was introduced in Chapter 1.3. In this chapter, we develop lower bounds for the zero-forcing number and upper bounds for the zero-forcing iteration index of a graph. In particular, I show that the branchwidth of a graph is a lower bound on the zero-forcing number, and I bound the zero-forcing number of a graph based on certain subgraphs, which I call *zero-forcing forts*. I show that the iteration index of a cubic graph is bounded above by $\frac{3}{4}$ the number of vertices in the graph. I also give a bound on the iteration index of a graph based on the number of claws and leaves contained in the graph.

This chapter is organized as follows. In Section 5.2, I bound the zero-forcing number from below using branchwidth, and in Section 5.3 I bound it from below using certain subgraphs, which I call *zero-forcing forts*. In Section 5.4, I provide the first non-trivial upper bound on the iteration index of cubic graphs, and I show that the iteration index of general graphs is inversely related to the number of disjoint vertex-induced claws in the graph.

*This chapter is adapted from [67].

5.2 A Branchwidth Bound on the Zero-Forcing Number

In this section, I show that the branchwidth of a graph, $bw(G)$, is a lower bound on the zero-forcing number. I recall the definition of a branch decomposition in Definition 5.1. I will again use rooted branch decompositions throughout this chapter.

Definition 5.1 (Robertson and Seymour [113])
Let G be a graph. A branch decomposition *of G is a pair consisting of a tree, T, such that every interior vertex of T has degree 3, and a bijection, τ, from the edges of G to the leaves of T. For a given edge, e, of T, the* middle set *of e is the set of all vertices of G that are incident to edges mapped by τ to leaves in both components of $T \backslash e$. The* width *of a given branch decomposition, (T, τ) is the maximum caridinality of the middle sets over all the edges of T. The branchwidth of G, $bw(G)$, is the minimum width over all branch decompositions of G.*

Barioli et al. [18] have already shown that tree-width, $tw(G)$, is a lower bound on the zero-forcing number. Robertson and Seymour [113] showed that $bw(G) \leq tw(G) + 1$. Thus, it is already known that $bw(G) \leq tw(G) + 1 \leq Z(G) + 1$. Our bound improves on this bound by 1 in the cases where $bw(G) = tw(G) + 1$.

Given a graph G, with a minimum forcing set Z, there may be a timestep in which two vertices can force the same vertex. Thus, there may be multiple ways in which the propagation of the infected vertices through the graph can occur. A *forcing chain*, $F = (f_1, f_2, ..., f_{|F|})$, of G given Z is an ordered set of vertices such that each vertex f_i in the chain forces f_{i+1}. Given a set, S, of iterations of the infection rule, a *maximal* forcing chain is a forcing chain that is not a proper subset of any other forcing chain that also uses the iterations of S. Let $\mathcal{F}$ be a set of

maximal forcing chains that force the graph G. Then, I call the pair $(Z, \mathcal{F})$ a *minimum forcing system* of G.

Our first lemma shows that every maximal forcing chain must start from the forcing set.

Lemma 5.1

Let G be a graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Let $F = (f_1, f_2, ..., f_{|F|-1}, f_{|F|})$ be a maximal forcing chain in $\mathcal{F}$. Then $f_1 \in Z$.

Proof: Suppose for contradiction that $f_1 \notin Z$. Then f_1 must be forced by some vertex. Also, since all of the subsequent vertices of F are forced after f_1, f_1 must be forced by some vertex not in F. Assume vertex v is the vertex that forces f_1. Since v forces f_1, v cannot be adjacent to any of the other vertices in F because otherwise v would have at least two unforced vertices and would not be able to force f_1. Then $F_v = (v, f_1, f_2, ..., f_{|F|-1}, f_{|F|})$ is a forcing chain that contains F. This contradicts the assumption that F was a maximal forcing chain. It follows that if F is a maximal forcing chain, then $f_1 \in Z$. ■

Theorem 5.1

Let G be a graph. Then, $Z(G) \geq bw(G)$.

Proof: Let $(Z, \mathcal{F})$ be a minimum forcing system of G. I will construct a branch decomposition with width at most $Z(G)$ in the following manner. Let $E(Z)$ be the set of edges of G that have both ends in the minimum forcing set Z. Let BD be a rooted branch decomposition of the graph induced by $E(Z)$, with bijection τ. Let M be any middle set of BD, and note that $|M| \leq |Z|$.

Now, if $E(Z)$ is not empty, then order the vertices in $V \backslash Z$ according to the order that they were forced in $(Z, \mathcal{F})$. For each of these vertices, v, in order, perform two steps, A and B. In step A, add a new root vertex, r, and a new leaf vertex, l, to

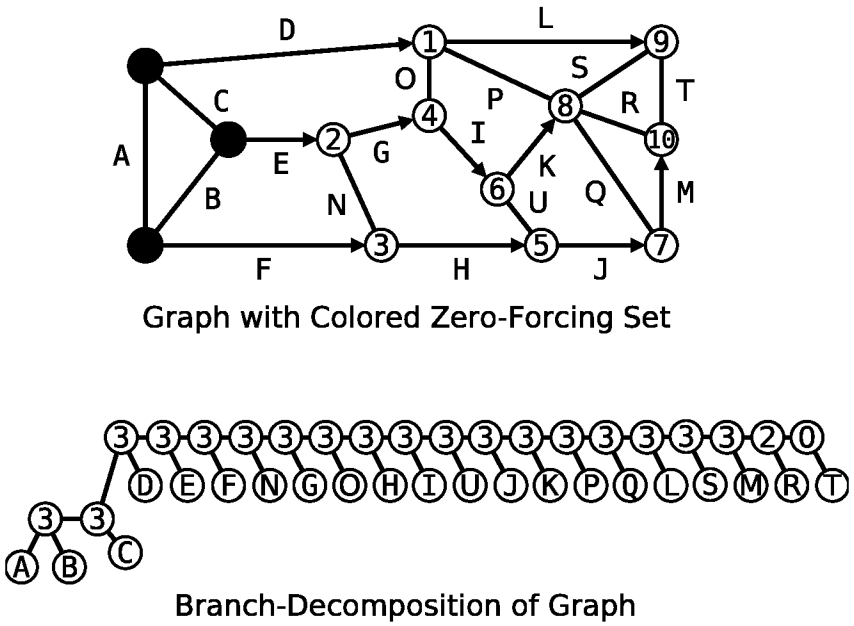


Figure 5.1 : Example of the construction of a branch decomposition from a zero-forcing system. Labels on the interior vertices of the branch decomposition tree give the width of those vertices. Note that the graph has $Z(G) = bw(G)$.

BD together with edges connecting r to the old root vertex and connecting r to l. Next, extend τ by mapping the leaf vertex l to the edge used to force v. Then, in step B, randomly order the edges that have v as one end and the other end either a vertex in Z or a vertex with lower order than v. For each of these edges, e, in order, add a new root and a new leaf vertex to BD. Again, extend τ by mapping the edge e to the new leaf vertex.

On the other hand, if $E(Z)$ is empty, then perform the same steps except that in the first step, the root and leaf vertex are the same. An example of our construction is given in Figure 5.1. It remains to show that (BD, τ) is a branch decomposition of G with width at most $Z(G)$.

First, I will show that BD is a rooted ternary tree. BD started out as a rooted ternary tree since it was a rooted branch decomposition of the graph induced by $E(Z)$, and except at the root, no edges incident to vertices of this original branch decomposition were added. Thus, none of the original vertices of BD can be part of a cycle. Since none of new vertices are connected to vertices that were added before them except for the single edges between the new root and the previous root, none of the added vertices can be part of a cycle either. Thus, BD is a tree. Since each vertex in BD that was added as a root is connected to the previous root, its leaf, and the subsequent root, each vertex added as a root has degree 3, except for the final root vertex and possibly the original root vertex. The final root vertex has degree 2, and the original root vertex may have degree 1 instead of 3 if it initially had no neighbors. Also, each vertex added as a leaf remains a leaf. Therefore, BD is a rooted ternary tree.

Since $(Z, \mathcal{F})$ was a minimum zero-forcing system of G, every vertex of G must be forced by $(Z, \mathcal{F})$. Therefore, every edge of G has a corresponding leaf added in some step of our process. Also, a leaf corresponding to some edge of G is added to BD only once. Thus, τ is a bijection from the leaves of BD to the edges of G. It follows that BD with τ is a rooted branch decomposition of G.

I now show that the width of BD is at most $Z(G)$. Since the original BD was a branch decomposition of the graph induced by $E(Z)$, it cannot have width greater than $|Z|$. So, consider the width of one of the vertices, r_e, that was added as a root. Let $e = \tau(l_e)$ where l_e is the leaf vertex that was added with the root vertex r_e. The root vertex r_e was added in either step A or step B.

Suppose r_e was added in step B. Then, any end of e must have either been in Z or incident to an edge added in a previous step A. Thus, each end of e must already

be in the middle set of a vertex added as a root in some previous step A. It follows that the largest middle set of a vertex added as a root in step B cannot be larger than the largest middle set of a vertex added as a root in step A.

On the other hand, suppose r_e was added in step A. Then, e must be used to force in $(Z, \mathcal{F})$. Let v be the end of e that forces the other end, w. All of the neighbors of v and v itself must be forced previous to w being forced. Thus, all the other edges incident to v must already have been added to the branch decomposition. Thus, v cannot be in the middle set of r_e. In particular, because vertices cannot be in the middle set after they force another vertex, each forcing chain $f \in F$ can have at most one element in the middle set of any vertex that was added as a root to BD in a step A. It follows that for any middle set M of BD, $|M| \leq |F|$, and by Lemma 5.1, $|F| \leq |Z|$. Therefore, $|M| \leq Z(G)$. ■

Theorem 5.1 is tight in the sense that there exist graphs that have $Z(G) = bw(G)$. One such graph is given in Figure 5.1. However, $Z(G)$ can also be much larger than $bw(G)$. Consider a star, that is, a tree with only one vertex that is not a leaf. For a star, $Z(G) = |V| - 2$, but $bw(G) = 1$. Thus, $Z(G)$ can be as much larger than $bw(G)$ as desired.

5.3 Subgraph Bounds on the Zero-Forcing Number

Now, I consider the distribution of the zero-forcing set throughout a graph by showing that certain subgraphs must contain a vertex in any forcing set. When these sets are disjoint, then their number obviously serves as a lower bound on $Z(G)$, but under certain conditions, I can still use their number as a lower bound even when the subgraphs are not disjoint. I start by defining a subgraph that must contain at least one element from the minimum forcing set.

Definition 5.2

Let $G = (V, E)$ be a graph and let $\emptyset \neq F \subset V$ be such that there does not exist a vertex in $V \backslash F$ that has exactly one neighbor in F. I will call F a zero-forcing fort *or simply a* fort *of G.*

Theorem 5.2

Let G be a graph, and let F be a fort of G. Then, any forcing set of G contains at least one vertex in F.

Proof: Suppose for contradiction that Z is a forcing set of G with $Z \cap F = \emptyset$. Let v be a vertex in F that is forced in the earliest iteration of the infection rule in which a vertex from F is forced. Note that v must exist since $F \neq \emptyset$. Since $F \cap Z = \emptyset$ and none of the vertices of F had been forced previous to the iteration in which v is forced, v must be forced by some vertex, w that is not in F. However, by definition of a fort, w must have at least two neighbors in F. Thus, w has at least two unforced neighbors in the iteration in which it forces v. This forcing contradicts the infection rule, and it follows that every forcing set contains at least one vertex in F. ■

Corollary 5.1

Let G be a graph, and let S be a set of pairwise disjoint forts in G. Then $Z(G) \geq |S|$.

Some examples of forts are given in Figure 5.2. In fact, $Z(G)$ is equal to the minimum number of vertices that intersect every fort. This fact follows from the observation that if a set of infected vertices cannot infect any more vertices, then the set of uninfected vertices forms a fort. Thus, if every fort is infected, then the infection must propagate until the entire graph is infected. This fact will be exploited in Section 6.2.2.

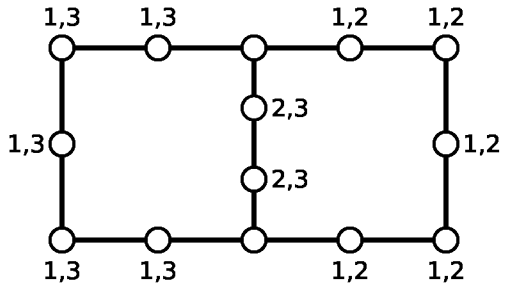


Figure 5.2 : An example of a graph that contains a family of forts, but only one disjoint fort. The set of nodes labeled with a 1, or the set of nodes labeled with a 2, or the set of nodes labeled with a 3 gives a fort of the graph, but the maximum size of any set of pairwise disjoint forts is 1. However, a more careful examination of the forts given by Corollary 5.2 shows that any set containing a vertex in every fort must have at least two vertices.

Corollary 5.1 provides a bound when forts are disjoint. However, there are certain graphs, for example the graph in Figure 5.2, that have multiple forts, but the maximum size of any set of pairwise disjoint forts in the graph is 1. Next, I give a theorem, Theorem 5.3, to create a bound from forts that are not necessarily disjoint.

To prove Theorem 5.3, I first prove Lemma 5.2 and its Corollary 5.2 that allows us to say that certain subgraphs contain forts. Next, I define *unpacked families*, which determine how the forts that I use to create my bound can intersect. Then, I prove Lemma 5.3, which shows that, when forts are given by Corollary 5.2, the forts form an unpacked family, and a vertex is removed from the graph, then a new unpacked family can be formed with at most one less member. The proof of Theorem 5.3 then uses Lemma 5.3 inductively to create a bound on the size of a minimum zero-forcing set.

Lemma 5.2

Let $G = (V, E)$ be a graph and let $H = (C, E_H)$ be a subgraph of G that has minimum degree at least 2. Let $C^B \subset C$ be the set of vertices of H that have neighbors in $V \backslash C$ in the graph G. If none of the vertices in C^B are adjacent in the graph H, then $F = C \backslash C^B$ is a fort of the graph G.

Proof:

$V \backslash F$ can be partitioned into $V \backslash C$ and C^B. Thus, for a given vertex $v \in V \backslash F$, either $v \in V \backslash C$ or $v \in C^B$. By definition of C^B, vertices in F are only adjacent to vertices in either F or in C^B. Therefore, if $v \in V \backslash C$, then v has no neighbors in F. On the other hand, if $v \in C^B$, then by the fact that H has minimum degree at least 2 and the fact that no two vertices in C^B are adjacent in H, v must have at least 2 neighbors in F. It follows that F is a fort of G. ■

Corollary 5.2

Let $G = (V, E)$ be a graph and let $H = (C, E_H)$ be a subgraph of G with minimum degree at least 2. Let $\emptyset \neq C^F \subset C$ be the set of vertices in C that have degree exactly 2 in G. Assume none of the vertices of $C \backslash C^F$ are adjacent in H. Then, C^F is a fort of G.

For the following proofs, it will be convenient to deal with various induced subgraphs of a graph $G = (V, E)$. To that end, I will use the following notation. For a subgraph $H = (C, E_H)$ of G, let C_G^F denote the set of vertices in C that have degree 2 in G. I will call graphs that have even degree at each vertex, but are not necessarily connected, *even-degree* graphs. For, two graphs $G = (V, E)$ and $H = (C, E_H)$, the symmetric difference of the two graphs, $G \triangle H$, is the graph induced by the symmetric difference of the edge sets of G and H, i.e. $G[E \triangle E_H]$. For a set $\mathcal{D}$ of graphs, the repeated symmetric difference of $\mathcal{D}$ is denoted by $\triangle \mathcal{D}$.

Definition 5.3

Let G be a graph and let $\mathcal{H}$ be a set of subgraphs of G such that no subgraph $H \in \mathcal{H}$ is the symmetric difference of some subset of subgraphs in $\mathcal{H} \backslash H$. Then, I call $\mathcal{H}$ an unpacked family *of subgraphs of G.*

Lemma 5.3

Let G be a graph, and let $\mathcal{H}$ be an unpacked family of vertex-induced even-degree subgraphs of G. For $H \in \mathcal{H}$, assume that no two vertices in C_G^F are adjacent in H. Let v be a vertex with degree 2 in G, and let R be the graph obtained by deleting v from G. Then, there is an unpacked family of even-degree subgraphs, $\mathcal{H}_R$, in R with $|\mathcal{H}_R| \geq |\mathcal{H}| - 1$, and for $H = (C, E_H) \in \mathcal{H}_R$, no two vertices in $C \backslash C_G^F$ are adjacent in G.

Proof: I will construct a family, $\mathcal{H}_R$, of subgraphs of R. For each subgraph $H \in \mathcal{H}$ such that H is also a subgraph of R, i.e. if $v \notin H$, I have $H \in \mathcal{H}_R$. I choose one subgraph $H_1 \in \mathcal{H}$ such that $v \in H_1$ so that H_1 does not exist in R. For each of the remaining subgraphs, $H_i \in \mathcal{H}$, such that $v \in H_i$, I add $H_1 \triangle H_i$ to $\mathcal{H}_R$. Note that $H_1 \neq H_i$ since $\mathcal{H}$ is an unpacked family; therefore, $H_1 \triangle H_i \neq \emptyset$. Since both H_1 and H_i are even-degree, $H_1 \triangle H_i$ is also even-degree. Also, since v has degree 2 and all of the subgraphs in $\mathcal{H}$ are even-degree subgraphs, both of the edges incident to v must be in both H_1 and H_i. Therefore, v is not in $H_1 \triangle H_i$, and $H_1 \triangle H_i$ is a subgraph of R. Note that after $H_1 \triangle H_i$ has been added to $\mathcal{H}_R$ for each $i \neq 1$, $|\mathcal{H}_R| \geq |\mathcal{H}| - 1$.

Now I show that $\mathcal{H}_R$ satisfies the requirement that for $H \in \mathcal{H}_R$, no two vertices in $C \backslash C_G^F$ are adjacent in G. For $H = (C, E_H) \in \mathcal{H}_R$, either $H \in \mathcal{H}$ or $H = H_1 \triangle H_i$ for two subgraphs $H_1, H_i \in \mathcal{H}$. If $H \in \mathcal{H}$, then by assumption, no two vertices in $C \backslash C_G^F$ are adjacent. On the other hand, suppose $H = H_1 \triangle H_i$ for two subgraphs $H_1 = (C_1, E_1), H_i = (C_i, E_i) \in \mathcal{H}$. Let h_1 and h_2 be two vertices in $C \backslash C_G^F$. By the

assumption on $\mathcal{H}$, h_1 and h_2 cannot be adjacent in either of H_1 and H_i. Therefore, they also cannot be adjacent in $H_1 \triangle H_i$.

It remains to show that $\mathcal{H}_R$ is an unpacked family. Suppose for contradiction that $\mathcal{H}_R$ is not an unpacked family. Then, there exists some subgraph $H \in \mathcal{H}_R$ that is the symmetric difference of some subset of subgraphs in $\mathcal{H}_R \backslash H$.

Case 1: Suppose $H \in \mathcal{H}$. Note that by the construction of $\mathcal{H}_R$, every subgraph in $\mathcal{H}_R \backslash H$ can be formed either as a subgraph from $\mathcal{H} \backslash H$ or as the symmetric difference of two subgraphs in $\mathcal{H} \backslash H$. Therefore, since H is the symmetric difference of some subset of subgraphs in $\mathcal{H}_R \backslash H$, H must also be the symmetric difference of some subset of subgraphs in $\mathcal{H} \backslash H$. However, this construction contradicts the assumption that $\mathcal{H}$ was an unpacked family.

Case 2: Suppose $H \notin \mathcal{H}$. Let H_2 be the subgraph in $\mathcal{H}$ such that $H = H_1 \triangle H_2$. Let $\mathcal{D} \subset \mathcal{H}_R \backslash H$ be the set of subgraphs whose symmetric difference is H.

Case 2.1: Suppose $\mathcal{D} \cap \mathcal{H} \neq \emptyset$. Let K be a subgraph in $\mathcal{D} \cap \mathcal{H}$. Since $H = \triangle \mathcal{D}$ and $K \in \mathcal{D}$, I have that $K = \triangle((\mathcal{D} \backslash K) \cup H)$. Thus, K is a subgraph in $\mathcal{H}_R$ that is also in $\mathcal{H}$, and K is the symmetric difference of sets in $\mathcal{H}_R \backslash K$. Therefore, this case is equivalent to case 1.

Case 2.2: Suppose $\mathcal{D} \cap \mathcal{H} = \emptyset$. By this supposition, each subgraph in $\mathcal{D}$ is of the form $H_1 \triangle H_i$ for some subgraph $H_i \in \mathcal{H}$ with $i \notin \{1, 2\}$. Let S be the set of all i such that $H_1 \triangle H_i \in \mathcal{D}$. Note that $H_2 \notin S$. Thus, I have

$$\underset{i \in S}{\triangle} (H_1 \triangle H_i) = H_2 \triangle H_1$$

By the associative and commutative properties of the symmetric difference, this expression is equivalent to either

$$H_1 \triangle (\underset{i \in S}{\triangle} H_i) = H_2 \triangle H_1$$

or

$$\triangle_{i \in S} H_i = H_2 \triangle H_1$$

depending on whether $|S|$ is odd or even, respectively. By taking the symmetric difference with H_1 on both sides, I get that either

$$\triangle_{i \in S} H_i = H_2$$

or

$$H_1 \triangle (\triangle_{i \in S} H_i) = H_2$$

depending on whether $|S|$ is odd or even, respectively. In either case, H_2 is the symmetric difference of sets in $\mathcal{H} \backslash H_2$. Therefore, $\mathcal{H}$ could not have been an unpacked family. From this contradiction, it follows that $\mathcal{H}_R$ must be an unpacked family.

Since $\mathcal{H}_R$ is an unpacked family with at least $|\mathcal{H}| - 1$ subgraphs, the lemma holds. ■

Theorem 5.3

Let G be a graph, and for a subgraph H of G, let H^F be the set of vertices in H that have degree 2 in G. Let $\mathcal{H}$ be an unpacked family of even-degree, vertex-induced subgraphs of G such that for $H \in \mathcal{H}$, none of the vertices in $H \backslash H_G^F$ are adjacent. Then $Z(G) \geq |\mathcal{H}| + 1$.

Proof: If G is acyclic then the theorem is trivial. If G contains a single even-degree subgraph, then G contains a cycle. Therefore, $Z(G) \geq 2$ and the theorem holds. So, assume $\mathcal{H} \geq 2$.

Case 1: Suppose the subgraphs $H \in \mathcal{H}$ are pairwise disjoint. Then, by Corollary 5.2, there is a fort of G corresponding to each subgraph $H \in \mathcal{H}$. Since the subgraphs are pairwise disjoint, the forts given by Corollary 5.2 must also be

pairwise disjoint, and by Lemma 5.2, any minimum forcing set contains at least one element from each fort. Thus, $Z(G) \geq |\mathcal{H}|$.

Now, the initial forcing set must have a vertex that is able to force. Thus, there must be a vertex that is in Z and also has all but one of its neighbors in Z. If this vertex is on an even-degree subgraph, then at least two vertices from that even-degree subgraph must be in Z. If the vertex is not on such a subgraph, then it is not part of one of the aforementioned forts of G. Thus, $Z(G) \geq |\mathcal{H}| + 1$.

Case 2: Suppose the subgraphs $H \in \mathcal{H}$ are not pairwise disjoint. Thus, the forts given by Corollary 5.2 are not necessarily disjoint. Let Z be a minimum forcing set and let v be a vertex in Z that is in more than one fort. Let R be the graph obtained by deleting v from G. Then by Lemma 5.3, there is an unpacked family of size $|\mathcal{H}| - 1$ in R. I can repeat this process to choose a set S with $|S| = |\mathcal{H}|$ and each vertex in S must be in Z.

Now, the initial forcing set must have a vertex that is able to force. Thus, there must be a vertex, v, that is in Z and also has all but one of its neighbors in Z. If $v \notin S$ then $|Z| \geq |\mathcal{H}| + 1$. If $v \in S$ then it has degree 2. So, v has a neighbor w that is also in Z. If w does not have degree 2, then it is not in one of the forts given by Corollary 5.2, and thus $|Z| \geq |\mathcal{H}| + 1$. If w does have degree 2, then any even degree subgraph that contains v also contains w and vise-versa. Thus, after either v or w is deleterd, the other cannot be part of a fort given by corollary 5.2. Consequently, it can be in at most one of the forts given by Corollary 5.2 and that fort must be the fort that contains v. Thus, again $|Z| \geq |\mathcal{H}| + 1$. ■

Although the assumptions of Theorem 5.3 may seem to be extremely restrictive, the theorem is useful for some very simple graphs. For example, for the graph in Figure 5.2, the maximum cardinality of a set of pairwise disjoint forts is 1. However,

the left and right cycles form an unpacked family of size 2 that satisfies the assumptions of the theorem. Thus, Theorem 5.3 shows that the zero-forcing number of the graph is at least 3, and for this example, the bound is tight.

5.4 Bounding the Iteration Index

In this section, I present bounds on the zero-forcing iteration index for cubic graphs, and I show that vertex-induced claws in a graph reduce its maximum possible iteration index. I will use the following terminology. A *maximum* forcing chain, F of G with respect to Z is forcing chain such that there does not exist a forcing chain D arising from a different sequence of forcing steps but the same initial forcing set Z with $|D| > |F|$. The *length* of a forcing chain is the number of edges in the forcing chain path. If a vertex is the only vertex forced in the iteration of the infection rule in which it is forced, then I call that vertex a *1-vertex*.

The main theorem of this section is Theorem 5.4, which states that a forcing set of a cubic graph must force the graph in no more than $\frac{3|V|}{4}$ iterations. Our strategy to prove this theorem is to bound the maximum possible number of iterations required to force a graph. Therefore, I first prove Lemma 5.4 to show that a forcing set must force at least one vertex in each iteration. Then, I prove Lemma 5.5, which shows that each forcing chain must start from a unique vertex in the forcing set. I will use Lemma 5.5 to prove that a certain number of vertices are in the forcing set and therefore don't require an iteration to force. Next, I prove Lemma 5.6, and use it in the proof of Theorem 5.4 to characterize the vertices in each forcing chain based on their neighbors. Lemmas 5.7, 5.8, 5.9, and 5.10 are all used in the proof of Theorem 5.4 to bound the maximum possible propagation time of a forcing set based on the characterization of the vertices in each forcing chain.

Lemma 5.4

Let G be a cubic graph and let $(Z, \mathcal{F})$ be a zero-forcing system of G. Then $(Z, \mathcal{F})$ forces at least one vertex in each iteration of the infection rule.

Proof: Suppose for contradiction that there exists an iteration, i, of the infection rule in which no vertex is forced. Then, no vertices of G were capable of forcing in iteration i. Since no vertices were forced in the iteration, no new vertices are capable of forcing in iteration $i + 1$. By induction, it follows that no vertices can be forced in any iteration after i. It follows that $(Z, \mathcal{F})$ must force at least one vertex in each iteration of the infection rule. ■

While I showed in Lemma 5.1 that forcing chains must start from the forcing set, the next lemma shows that these starting vertices must be unique to each forcing chain.

Lemma 5.5

Let G be a graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Let $F = (f_1, f_2, ..., f_{|F|-1}, f_{|F|})$ and $D = (d_1, d_2, ..., d_{|D|-1}, d_{|D|})$ be two distinct maximal forcing chains in $\mathcal{F}$. Then $f_1 \neq d_1$.

Proof: Suppose for contradiction that $f_1 = d_1$. Since F and D are maximal, $F \not\subset D$ and $D \not\subset F$. Since I also have $F \neq D$, there must exist some $1 < i \leq |F|$ such that $f_i \neq d_i$. Let i be the minimum such index. Then, $f_{i-1} = d_{i-1}$. Since both F and D are forcing chains, f_{i-1} must be capable of forcing both f_i and d_i. However, f_i can only be capable of forcing if it only has one unforced neighbor. Thus, both f_i and d_i cannot be unforced, and f_{i-1} cannot force both f_i and d_i. It follows that $f_1 \neq d_1$. ■

The next lemma shows that vertices in a forcing chain cannot be adjacent to other vertices in the chain that are not either immediately before or after in the forcing chain.

Lemma 5.6

Let G be a cubic graph and let $(Z, \mathcal{F}$ be a minimum zero-forcing system of G. Let $F = (f_1, f_2, ..., f_{|F|-1}, f_{|F|})$ be a forcing chain in $\mathcal{F}$. Then for $i \in \{1...|F|\}$, f_i is not adjacent to any $f_j \in F$ with $j \notin \{i-1, i+1\}$.

Proof: Suppose for contradiction that there exists an i such that f_i is adjacent to $f_j \in F$ and $j \notin \{i-1, i+1\}$. Consider the minimum such i. Since i is minimum, $j > i+1$. Thus, f_i is adjacent to both f_{i+1}, since f_i forces f_{i+1}, and f_j. However, since $j > i+1$ and f_j is part of F, f_j cannot be forced until after f_{i+1} is forced. Since f_i is adjacent to f_j, f_i cannot force f_{i+1} until after f_j is forced. However, in the forcing chain F, f_i forces f_{i+1} before f_j is forced. From this contradiction, it follows that there does not exist an i such that f_i is adjacent to $f_j \in F$ and $j \notin \{i-1, i+1\}$. ■

Definition 5.4

Let G be a graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Number the iterations of the infection rule consecutively starting from 0. I will call the iteration in which a vertex v is forced the forcing time *of v, and I will denote its number by $T(v)$.*

In our consideration of forcing chains, it will be convenient to name the different parts of the forcing chains. Consequently, for a maximal forcing chain $F = (f_1, f_2, ..., f_{k-1}, f_k)$, I will call f_1 the *start* vertex, f_k the *end* vertex, and all other vertices *middle* vertices of the forcing chain. I can partition the vertices of a graph into sets based on their place in their own forcing chain and their relation to adjacent forcing chains. Note that Definition 5.5 does not require the graph to be cubic.

Definition 5.5

Let G be a graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Let M be the set of all interior vertices of forcing chains in $\mathcal{F}$. Let S be the set of start vertices of forcing chains with length at least 1, and let E be the set of end vertices of forcing chains with length at least 1. Let L be the set of vertices of forcing chains with length 0. Let E_i (likewise S_i, M_i, L_i) be the set of vertices in E (S, M, L) that are adjacent to i vertices in M that are in a different forcing chain. Let S_1^S be the set of vertices in S_1 that are adjacent to a vertex in either S or L, and let S_1^E be the set of vertices in S_1 that are adjacent to a vertex in E and in a different forcing chain.

It is easy to see from Figure 5.3(a) that a vertex in S_2 and one of its neighbors must force in the same iteration. This observation leads to the following lemma.

Lemma 5.7

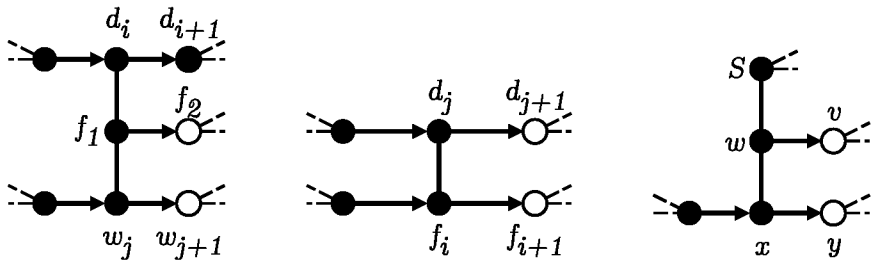
Let G be a cubic graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Let $F = (f_1, f_2, ..., f_{|F|})$, $D = (d_1, d_2, ..., d_{|D|})$, and $W = (w_1, w_2, ..., w_{|W|})$ be forcing chains, not necessarily unique, in $\mathcal{F}$. If f_1 is adjacent to interior vertices d_i and w_j of D and W, then f_2 is forced in the same iteration as either d_{i+1} or w_{j+1}.

Proof: Assume without loss of generality that $T(d_i) \leq T(w_j)$. By the definition of zero-forcing, f_1 cannot force f_2 until both d_i and w_j are forced. Thus, $T(f_2) = T(w_j) + 1$. However, at $T(w_j)$, w_{j-1} and f_1 are already forced. So, w_{j+1} must be forced in the next step and $T(w_{j+1}) = T(w_j) + 1$. Since I have shown that $T(f_2) = T(w_{j+1})$, f_2 is forced in the same iteration as w_{j+1}, and the result follows. ■

Likewise, Figure 5.3(b) shows that each pair of vertices in M_1 must force in the same iteration, and I again obtain a lemma.

Lemma 5.8

Let G be a cubic graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Let



(a) (b) (c)

Figure 5.3 : Examples of S_2, M_1, and S_1^S vertices. In (a), as soon as both neighbors of the S_2 vertex, f_1, are forced, then it is capable of forcing, but one of its neighbors is also capable of forcing in the same iteration. The gray vertex may be forced at any iteration up to the iteration in which the S_2 vertex is able to force. In (b), as soon as both vertices in M_1 are forced, then they are both capable of forcing, but they are both not capable of forcing until that iteration. In (c), as soon as the neighbor, x, in M is forced, x and the S_1^S vertex, w, are both capable of forcing.

$F = (f_1, f_2, ..., f_{|F|})$ and $D = (d_1, d_2, ..., d_{|D|})$ be distinct forcing chains in $\mathcal{F}$. If, some vertex $f_i \in F$ for $1 < i < |F|$ is adjacent to a vertex $d_j \in M$ for $1 < j < |D|$, then f_{i+1} and d_{j+1} are forced in the same iteration of the infection rule.

Proof: Neither d_j nor f_i can force the next vertex in their respective chains until both of them are forced since they will either have two unforced neighbors or be unforced themselves up to that point. When both d_j and f_i are forced, then they are both capable of forcing the next vertex in their respective chains. Thus, the forcing step that forces f_{i+1} must also force d_{j+1}. ■

I can obtain another lemma from Figure 5.3(c), which shows that a vertex in S_1^S must force in the same iteration as one of its neighbors.

Lemma 5.9

Let G be a cubic graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Let v be a vertex that is preceded in its forcing chain by a vertex, $w \in S_1^S$, and let x be the vertex in M that is adjacent to w. Then, v is forced in the same iteration as the vertex that follows x in its forcing chain.

Proof: Note that since $x \in M$, it must have a vertex following it in its forcing chain. Since $w \in S_1^S$, it only has two unforced neighbors in the initial iteration, namely v and x. Thus, w is capable of forcing as soon as x is forced. The vertex preceding x in its forcing chain must be forced before x, and w is in Z. Thus, x is capable of forcing as soon as x is forced. Therefore, both w and x must force the subsequent vertices in their respective forcing chains in the same iteration. ■

Lemma 5.10

Let G be a cubic graph and let $(Z, \mathcal{F})$ be a minimum forcing system of G. Then $|E_1| + 2|E_0| \geq |S_1^E|$.

Proof: By definition of S_1^E, each vertex in S_1^E must be adjacent to a vertex in E. Let w be a vertex in E. Since G is cubic, w has three neighbors. Since $w \in E$, one of these neighbors must be the preceding vertex in the forcing chain that contains w. Now w is in exactly one of E_2, E_1, or E_0. If w is in E_2, then it is adjacent to two vertices that are in M and not from the same forcing chain as w. Therefore, w cannot be adjacent to a vertex from S_1^E. If w is in E_1, then it is adjacent to exactly one vertex that is in M and not from the same forcing chain as w. Therefore, w can be adjacent to at most one vertex from S_1^E. Finally, if w is in E_0, then w can be adjacent to at most two vertices from S_1^E. It follows that $|E_1| + 2|E_0| \geq |S_1^E|$. ■

With the previous lemmas, I am ready to prove my bound on the iteration index of cubic graphs.

Theorem 5.4

Let $G = (V, E)$ be a cubic graph, and let $(Z, \mathcal{F})$ be a forcing system of G. Then $(Z, \mathcal{F})$ forces G in at most $\frac{3|V|}{4}$ iterations.

Proof: Consider a cubic graph G with a forcing system $(Z, \mathcal{F})$. Let $F = (f_1, f_2, ..., f_{|F|})$ be a maximal forcing chain in $\mathcal{F}$. Since G is cubic, then by Lemma 5.6, for each interior vertex $f_i \in F$, there exists a vertex $w_i \notin F$ such that w_i is adjacent to f_i. There are four possibilites for these vertices.

- $w_i \in L$. That is, $w_i \in Z$ and w_i is part of a maximal forcing chain of length 0.
- $w_i \in S$. That is, $w_i \in Z$ and w_i is the start vertex of some maximal forcing chain $W = (w_1, ..., w_{|W|})$ with length at least 1.
- $w_i \in M$. That is, w_i is an interior vertex of some maximal forcing chain $W = \{w_1, ..., w_{|W|}\}$.

- $w_i \in E$. That is, w_i the end vertex of some maximal forcing chain $W = \{w_1, ..., w_{|W|}\}$ with length at least 1.

First, I will count the number of forcing iterations required to force G. Let T be the number of iterations required for $(Z, \mathcal{F})$ to force G. The only vertices that will be forced are the middle and end vertices of forcing chains. I can identify these vertices by the type of vertex that forces them. Middle and end vertices must be forced by either middle or start vertices of their forcing chain. By Lemma 5.4, a vertex must be forced in each iteration. Therefore, $T \leq |S_0| + |S_1| + |S_2| + |M_1| + |M_0|$.

From Lemma 5.7, I know that each middle or end vertex, f_2, that is preceded in its forcing chain by a vertex, $f_1 \in S_2$, is forced in the same iteration as another vertex, w_{j+1}, that is preceded by a vertex, $w_j \in M_0$ with w_j adjacent to f_1 (see Figure 5.3(a)). Therefore, I only need to consider one iteration for the pair of f_2 and w_{j+1}. Therefore, each vertex preceded by a vertex in S_2 is paired with a vertex that is preceded by a vertex in M_0. This observation allows us to reduce the maximum possible number of iterations by $|S_2|$. So I have,

$$T \leq |S_0| + |S_1| + |M_1| + |M_0|$$

From Lemma 5.8 I have that each pair of vertices in M_1 force the next vertex in their respective chains in the same iteration. So, I only need to consider one iteration for each such pair. This consideration gives

$$T \leq |S_0| + |S_1| + \frac{1}{2}|M_1| + |M_0|$$

Since G is cubic, and Lemma 5.6 prevents a vertex from being adjacent to more than two vertices in its own forcing chain, each vertex in M is adjacent to a unique vertex from a different forcing chain. For the vertices in M_0, the adjacent vertex

cannot be from M. Therefore,

$$|M_0| = 3|L_3| + 2|L_2| + |L_1| + 2|S_2| + |S_1| + 2|E_2| + |E_1|$$

. Using the above identity gives

$$T \leq 2|S_2| + 2|S_1| + |S_0| + \frac{1}{2}|M_1| + 3|L_3| + 2|L_2| + |L_1| + 2|E_2| + |E_1|$$

Since, by Lemma 5.5, each maximal forcing chain of length at least 1 must contain one start vertex and one end vertex, I have that
$|E_2| + |E_1| + |E_0| = |S_2| + |S_1| + |S_0|$. I use this identity to get

$$T \leq |S_2| + |S_1| + \frac{1}{2}|M_1| + 3|L_3| + 2|L_2| + |L_1| + 3|E_2| + 2|E_1| + |E_0|$$

I can also partition S_1 into S_1^E and S_1^S to get

$$T \leq |S_2| + |S_1^E| + |S_1^S| + \frac{1}{2}|M_1| + 3|L_3| + 2|L_2| + |L_1| + 3|E_2| + 2|E_1| + |E_0|$$

Now from Lemma 5.9, I have that a vertex, v, that is preceded by a vertex, w, in S_1^S is forced in the same iteration as a vertex, y, that follows a vertex $x \in M_0$ with x adjacent to w (see Figure 5.3(c)). Thus, I only need to consider one iteration for the pair of v and y. Note that these pairs are different from the previous pairs that I obtained from Lemma 5.7 because the vertex in M_0 that precedes y must be adjacent to a vertex in S_1 whereas the vertex that precedes w_{j+1} in the pair from Lemma 5.7 was adjacent to a vertex in S_2 (see Figure 5.3(a)). This observation allows us to reduce the maximum possible number of iterations by $|S_1^S|$. Thus I have,

$$T \leq |S_2| + |S_1^E| + \frac{1}{2}|M_1| + 3|L_3| + 2|L_2| + |L_1| + 3|E_2| + 2|E_1| + |E_0|$$

Now, I count the number of vertices in G. Since each vertex of G is in one of the sets L, S, M, or E, the number of vertices in G is

$$|V = |L_3| + |L_2| + |L_1| + |L_0| + |S_2| + |S_1| + |S_0| + |M_1| + |M_0| + |E_2| + |E_1| + |E_0|$$

Using the identity,

$$|M_0| = 3|L_3| + 2|L_2| + |L_1| + 2|S_2| + |S_1| + 2|E_2| + |E_1|$$

I get that

$$|V| = 4|L_3| + 3|L_2| + 2|L_1| + |L_0| + 3|S_2| + 2|S_1| + |S_0| + |M_1| + 3|E_2| + 2|E_1| + |E_0|$$

Now, using the identity $|E_2| + |E_1| + |E_0| = |S_2| + |S_1| + |S_0|$ gives

$$|V| = 4|L_3| + 3|L_2| + 2|L_1| + |L_0| + 2|S_2| + |S_1| + |M_1| + 4|E_2| + 3|E_1| + 2|E_0|$$

By Lemma 5.10, I have that $|E_1| + 2|E_0| \geq |S_1^E|$. Partitioning S_1 into S_1^E and S_1^S and using Lemma 5.10, I have that

$$|V| \geq 4|L_3| + 3|L_2| + 2|L_1| + |L_0| + 2|S_2| + \frac{4}{3}|S_1^E| + |S_1^S| + |M_1| + 4|E_2| + \frac{8}{3}|E_1| + \frac{4}{3}|E_0|$$

Taking the ratio of T to V, I have that $\frac{T}{V} \leq$

$$\frac{3|L_3| + 2|L_2| + |L_1| + |S_2| + |S_1^E| + \frac{1}{2}|M_1| + 3|E_2| + 2|E_1| + |E_0|}{4|L_3| + 3|L_2| + 2|L_1| + |L_0| + 2|S_2| + \frac{4}{3}|S_1^E| + |S_1^S| + |M_1| + 4|E_2| + \frac{8}{3}|E_1| + \frac{4}{3}|E_0|}$$

For each term in the above expression, the coefficient in the denominator is at least $\frac{4}{3}$ the coefficient of the term in the numerator. This relationship gives the result, $\frac{T}{|V|} \leq \frac{3}{4}$. ■

Corollary 5.3

Let G be a cubic graph. Then, $I(G) \leq \frac{4|V|}{3}$.

Figure 5.4 shows that the bound of Theorem 5.4 is asymptotically tight. However, Corollary 5.3 is not necessarily tight since there may be minimum forcing sets that force the graph in fewer iterations. In other words, there may be minimum forcing sets that force the graph in less iterations than the forcing set that uses the most iterations.

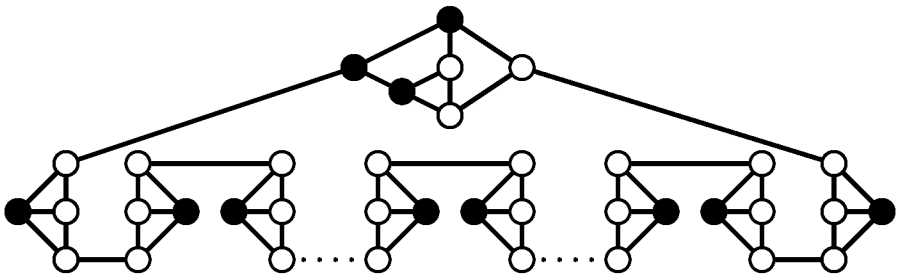


Figure 5.4 : An example of a graph for which Theorem 5.4 is tight. The number of iterations required by the minimum forcing set given by the colored vertices approaches $\frac{3|V|}{4}$ as the structure on the bottom of the graph is repeated.

Now, I shift my attention to general graphs and the relationship between the iteration index and certain subgraphs called claws. Given a graph G, a *claw* is just a vertex-induced complete bipartite graph $K_{1,3}$.

Theorem 5.5

Let G be a graph, and let K be a set of disjoint claws in G. Then $I(G) \leq |V| - \frac{1}{2}|K|$.

Proof: Let $(Z, \mathcal{F})$ be a minimum forcing system of G. By Lemma 5.6, at most three of the vertices in the claw are in the same forcing chain. There are two possible cases: either all vertices in the claw are in M, or the claw contains a vertex in L, S, or E. If all vertices in the claw are in M, then the claw contains two adjacent middle vertices from different forcing chains and by Lemma 5.8, the vertices that come after these two middle vertices in their respective chains must be forced in the same timestep. Therefore, the maximum possible iteration index of the graph must be reduced by one for each such pair of adjacent middle vertices.

On the other hand, if there is at least one start or end vertex in the claw (vertices in L are considered to be both start and end vertices), let K_2 be the set of claws that contain a start or end vertex. Since each start and end vertex belongs to

exactly one forcing chain, there are at least $\frac{1}{2}|K_2|$ forcing chains in G. By Lemma 5.5, there is a distinct member of Z for each forcing chain. Thus, the maximum possible number of vertices that are not in Z is $|V| - \frac{1}{2}|K_2|$. Now, let K_1 be the set of claws that contain four vertices in M. For each such claw, there is a pair of vertices that are forced in the same iteration, and since the claws are disjoint, these pairs are unique to each claw. It follows that the maximum possible number of iterations required to force G is $|V| - |K_1| - \frac{1}{2}|K_2|$. Since $K_1 \cap K_2 = \emptyset$, I have that $|K| = |K_1| + |K_2|$. It follows that $I(G) \leq |V| - \frac{1}{2}|K|$. ■

Leaves of a graph also allow a better bound on both the forcing number and the iteration index.

Theorem 5.6

Let G be a graph, and let C be the set of vertices with degree 1 in G. Then $Z(G) \geq \frac{1}{2}|C|$.

Proof: Let $(Z, \mathcal{F})$ be a minimum forcing system of G, and let v be a vertex with degree 1 in G. Then, since v has degree 1, it is either in Z or is an end vertex of a forcing chain in $\mathcal{F}$. Therefore, $|\mathcal{F}| \geq \frac{1}{2}|C|$. By Lemma 5.5, each of the forcing chains in $\mathcal{F}$ contains a distinct vertex in Z. Thus, $Z(G) \geq \frac{1}{2}|C|$. ■

Corollary 5.4

Let $G = (V, E)$ be a graph, and let C be the set of vertices with degree 1 in G. Then $I(G) \leq |V| - \frac{1}{2}|C|$.

I can also combine Theorem 5.5 and Corollary 5.4.

Theorem 5.7

Let $G = (V, E)$ be a connected graph, let K be a set of disjoint claws in G, and let C be the set of vertices with degree 1 in G and such that $C \cap K = \emptyset$. Then, $I(G) \leq |V| - \frac{1}{2}|C| - \frac{1}{2}|K|$.

Proof: Let $(Z, \mathcal{F})$ be a minimum forcing system of G. Let K_2 be the set of claws that contain a start or end vertex and let K_1 be the set of claws that contain four vertices in M. Since each claw in K_2 and each leaf contains either a start or end vertex and the claws and leaves are disjoint, there must be at least $\frac{1}{2}(|K_2| + |C|)$ forcing chains in $\mathcal{F}$. It follows that the maximum possible number of vertices that are not in Z, and therefore the maximum possible number of iterations required to force G, is $|V| - \frac{1}{2}(|C| + |K_2|)$. Again by the same reasoning that was used in the proof of Theorem 5.5, there must be a distinct pair of vertices that are forced in the same iteration for each member of K_1. This fact reduces the maximum possible number of iterations required to $|V| - \frac{1}{2}|C| - \frac{1}{2}|K_2| - |K_1|$. Since $|K| = |K_1| + |K_2|$, I have that $I(G) \leq |V| - \frac{1}{2}|C| - \frac{1}{2}|K|$. ■

5.5 Conclusion

In this chapter, I presented bounds for the zero-forcing number and zero-forcing iteration index of a graph. For the zero-forcing number, I showed that the branchwidth of a graph is a tight lower bound on the zero-forcing number of the graph. This bound complements the treewidth bounds of Barioli et al. [18]. I also showed that any zero-forcing set must contain a vertex in every zero-forcing fort. Thus, if the forts are disjoint, the number of forts provides a bound on the zero-forcing number. I will exploit the fort theory in the next chapter to build integer programming formulations for the zero-forcing problem.

For the zero-forcing iteration index, I showed that any forcing set of a cubic graph forces the graph in at most $\frac{3|V|}{4}$ iterations, where $|V|$ is the number of vertices in the graph. This bound is asymptotically tight; however, there may be other minimum forcing sets of the graph that force the graph in fewer iterations. Thus,

the iteration index, $I(G)$, of a cubic graph is at most $\frac{3|V|}{4}$, but it is not known whether this bound is tight. I also showed that disjoint claws and leaves in a graph reduce its maximum possible iteration index. Cubic graphs do not have leaves, and they can have at most $\frac{|V|}{4}$ disjoint claws. Thus, the bound of Corollary 5.3 is at least as good as the bound based on disjoint claws; however, the claw bound applies to any graph, not just cubic graphs.

Chapter 6

Computing Minimum Zero-Forcing Sets*

This chapter examines computational methods for finding minimum cardinality zero-forcing sets. The chapter is organized as follows. First, I give a description of the Wavefront algorithm of Butler et al. [35], which is the current state-of-the-art method in zero-forcing computatation. I show that the Wavefront algorithm is correct and give a bound on its worst-case memory requirements. To my knowledge, neither complexity nor correctness of the Wavefront algorithm has been addressed in the literature. Second, I derive integer programming formulations for the zero-forcing problem and compare them to each other and to Wavefront. The first formulation strategy is based on the times at which vertices are forced, but the second formulation strategy is based on generating violated forts. Thus, both strategies use results from Chapter 5.

6.1 Wavefront Algorithm

In this section, I give a description of the Wavefront algorithm [35]. I also prove that it is correct in Theorem 6.1 and give a result about its worst-case memory requirements in Theorem 6.2. It will be convenient to define the *derived set* of a set of infected vertices. The derived set of a set, S, will be denoted by $zfs(S)$ and is the set of all vertices that will become infected using S as an initial infected set. As

*This chapter is adapted from [27]

has been noted by the AIM Minimum Rank – Special Graphs Work Group [3], $zfs(S)$ is uniquely determined by S. This can easily be seen by noting that the set of vertices that are not forced are precisely those vertices that are contained in some fort that does not intersect S.

Algorithm 6.1: Wavefront Algorithm [35]

Data: $G = (V, E)$
Result: Size of minimum forcing set of G
$\mathcal{C} = \{(\emptyset, 0)\}$;
for $R \in \{1, 2, ..., N\}$ **do**
  for $(S, r) \in \mathcal{C}$ **do**
    for $v \in G$ **do**
      $k =$ number of unforced neighbors of v;
      $C = zfs(S \cup \{v\} \cup N(v))$;
      if $(C, i) \notin \mathcal{C}$ *for* $i \leq R$ **then**
        if $v \notin S$ *and* $k \leq R - r$ **then**
          Add $(C, r + k)$ to $\mathcal{C}$;
        end
        if $v \in S$ *and* $k - 1 \leq R - r$ **then**
          Add $(C, r + k - 1)$ to $\mathcal{C}$;
        end
      end
    end
  end
  if $(V, z) \in \mathcal{C}$ **then**
    Return z;
  end
end

Recall Definition 5.2 of a fort from Chapter 5. A fort is a non-empty set of vertices such that no vertex outside the fort is adjacent to exactly one vertex in the fort. To prove the correctness of the Wavefront algorithm, I first need to prove Lemma 6.1 concerning the existence of forts inside other forts.

Lemma 6.1

Let F be a fort of a graph G and let $N[F]$ be the set containing F and all the neighbors of vertices in F. Let S be a set that does not contain some vertex $v \in N[F]$ and all but one of the vertices in $N(v) \cap F$. Then $F \backslash S$ is a fort of G.

Proof: Since F was a fort, every vertex in $N[F]$ is either in F or has at least two neighbors in F. Let w be a vertex in $N[F \backslash S]$. Then, w must be in $N[F]$. Since S did not contain some vertex in $N[F]$ and all but one of its neighbors, w must either not be in S or w must have at least two neighbors that are in F but not in S. Thus, w must either be in $F \backslash S$ or w must have at least two neighbors in $F \backslash S$. Thus, $F \backslash S$ is a fort of G. ■

Theorem 6.1

The Wavefront algorithm returns a minimum zero-forcing set.

Proof: For a set $H \subset V$, let Z_H be a zero-forcing set that has minimum size subject to the constraint that it contains H as a subset. Suppose $H_0 = \emptyset$, then by Lemma 6.1, Z_{H_0} must contain some vertex, v_1 of G and all but one of its neighbors. This set, call it H_1, must be added to $\mathcal{C}$ by step $|N(v_1)|$ of the Wavefront algorithm. Now, again by Lemma 6.1, Z_{H_1} must contain the members of H_2, which contains some vertex v_2 and all but one of its neighbors, that are not in $zfs(H_1)$. The uninfected members of H_2, call this set H_2^U must be added to H_1 by step $|N(v_1) \cup H_2^U|$ of the wavefront algorithm. This process can be repeated as necessary until $zfs(N(v_1) \cup H_2^U \cup ... \cup H_i^U) = V$. By Lemma 6.1, $|Z(G)| = |Z_{H_0}| = |Z_{H_1}| = ... = |Z_{H_i}|$. Therefore, the Wavefront Algorithm returns a minimum zero-forcing set. ■

Theorem 6.2

For a graph with N vertices and zero-forcing number z, at any step, s, of the

Wavefront algorithm,

$$|\mathcal{C}| \leq \sum_{i=1}^{s} \frac{N!}{(N-i)!i!}$$

.

Proof: Wavefront does not add multiple sets that have the same closure to $\mathcal{C}$. Since each permutation of a set has the same closure as the other permutations, the maximum number of sets added to $\mathcal{C}$ by step s, is the number of combinations of vertices with size at most s. This number of combinations is $\sum_{i=1}^{s} \frac{N!}{(N-i)!i!}$. ■

Note that Theorem 6.2 is a worst-case bound. Although the Wavefront algorithm in the worst case is no better than enumerating all possible subsets, the Wavefront algorithm performs much better than the worst case bound when the closure of vertex subsets is larger than the starting subset. This improvement comes from some vertices being forced and having no unforced neighbors, and therefore never being a possible choice to add to the sets in $\mathcal{C}$. The worst case performance of Wavefront is realized in graphs with only isolated vertices (vertices that have no neighbors), but since these vertices must be in any forcing set, the graph can be preprocessed to remove all isolated vertices before Wavefront is run. However, stars (trees with only one interior vertex) also lead to very poor complexity. Since any two leaves of the star form a fort, Wavefront is required to check every combination of leaves of size less than $N-2$ before it will find a forcing set.

6.2 Integer Programming Methods

This section describes integer programming formulations and solution strategies for the zero-forcing problem. The integer programming formulations presented here come from two distinct perspectives on the zero-forcing problem. The first

perspective is straightforward and tries to model zero-forcing as a dynamic graph infection process. This approach must consider the time that each vertex is forced and uses the vertices forced at each timestep to determine the vertices that can be forced in the next timestep. The second perspective uses the theory of zero-forcing forts from Chapter 5. This approach sees zero-forcing not as a dynamic graph infection process, but rather as a type of fort covering problem.

6.2.1 Infection Perspective

I now give the formulation for zero-forcing as a dynamic process. In the formulation, the graph is viewed as a directed graph. Each edge of the initial undirected graph is replaced by two edges giving both possible directions of that edge. Let V be the vertex set of the graph, E the edge set of the directed graph and T the maximum number of timesteps required to force the graph. I use three sets of variables. There is a binary variable, s, for each vertex that indicates whether the corresponding vertex is in the forcing set. Also, for each vertex, there is an integer variable, x, between 0 and T that indicates the iteration of the zero-forcing infection rule in which the corresponding vertex is forced. Finally, there is a binary variable, y, for each directed edge that indicates whether the tail of the corresponding edge forces the head of the edge. For an edge e and vertex v, I use the notation $e \to v$ to indicate that v is the head of e.

Model 6.1 Integer Program Model of the Zero-Forcing Problem based on Infection

$$\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} s_v \\
\text{subject to:} \quad & s_v + \sum_{e \to v} y_e = 1 && \forall v \in V && (1) \\
& x_u - x_v + (T+1)y_e \leq T && \forall e = (u,v) \in E && (2) \\
& x_w - x_v + (T+1)y_e \leq T && \forall e = (u,v) \in E, \forall w \in N(u) - v && (3) \\
& x \in \{0, 1, ..., T\} \\
& s \in \{0, 1\} \\
& y \in \{0, 1\}
\end{aligned}$$

Theorem 6.3

The optimum of Model 6.1 is equal to the size of a minimum zero-forcing set.

Proof: Consider a zero-forcing system $(Z, \mathcal{F})$ (zero-forcing systems were defined in Section 5.2) of a graph G. Every vertex of G must be forced. Therefore, every vertex, v, of G is either in Z (i.e. $s_v = 1$) or is forced by some other vertex of G (i.e. $y_e = 1$ and v is the head of e). Thus, constraint (1) must be satisfied. Now, let x_v be the iteration in which v is forced by $(Z, \mathcal{F})$. Since a vertex cannot force until all but one of its neighbors are forced, we have that for every edge, $e = (v, w)$ for which $y_e = 1$, it must be that v is forced before w and thus $x_v < x_w$. Likewise $x_i < x_w$ for all neighbors i of v. Thus, constraints (2) and (3) are satisfied. If $y_e = 0$, then constraints (2) and (3) are satisfied since T is the maximum difference between the forcing times of two vertices. Thus, the constraints are valid for any zero-forcing system.

Now, let (s, x, y) be a solution of Model 6.1 for a given graph G. Then let Z be the set of all vertices for which $s_v = 1$. Let $\mathcal{F}$ be the set of paths induced by the edges for which $y_e = 1$. For any edge $e = (v, w)$ for which $y_e = 1$, by constraints (2) and (3) it must be the case that v can force w through some number of applications

of the zero-forcing rule. Since we also have that every vertex is either in Z or has an incoming edge with $y_e = 1$ it follows that every vertex in G eventually is forced. Therefore, $(Z, \mathcal{F})$ is a zero-forcing system of G. The result follows. ■

Model 6.1 has several nice features. It not only finds the minimum zero-forcing number and a minimum zero-forcing set, but it also gives the paths that the zero-forcing infection takes through the graph. Also, there is a polynomial number of constraints and variables relative to the graph size; therefore, column and row generation is not needed. However, the downfall of this model is its reliance on constraints (2) and (3) which are of a big-M form. Even though we have worst-case bounds on the size of T from Section 5.4, the big-M constraints still lead to poor performance.

6.2.2 Fort Covering Perspective

My next formulation has no big-M constraints. In Model 6.2, the variables s_v again indicate whether vertex v is in the zero-forcing set. The set $\mathcal{B}$ is the set of all forts of the given graph (recall that forts were defined in Definition 5.2).

Model 6.2 Integer Program Model of the Zero-Forcing Problem based on Forts

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} s_v \\ \text{subject to:} \quad & \sum_{v \in B} s_v \geq 1 \qquad \forall B \in \mathcal{B} \qquad (1) \\ & s \in \{0, 1\} \end{aligned}$$

Theorem 6.4

The optimum of Model 6.2 is equal to the size of a minimum zero-forcing set.

Proof: Let s be a solution of Model 6.2 for a given graph $G = (V, E)$, and let Z be the set of all vertices v for which $s_v = 1$. Suppose for contradiction that Z is not a

zero-forcing set of G. Then, let $zfs(Z)$ be the derived set of Z. Since Z is not a forcing set of G, $zfs(Z) \neq V$. Therefore, $V \backslash zfs(Z)$ is a fort of G that does not contain a vertex in Z. However, this case requires that constraint (1) of Model 6.2 be violated. It follows that Z must be a zero-forcing set of G.

Now, let Z be a zero-forcing set of G. Since $zfs(Z) = V$, G cannot contain a fort that does not contain some element of Z. Therefore, Z is a feasible solution of Model 6.2. The result follows. ■

In contrast to Model 6.1, Model 6.2 has the advantage of having no big-M type constraints. However, unlike Model 6.1, it does not find the paths of the zero-forcing process. The most important issue with Model 6.2 is that there are potentially exponentially many forts in a given graph. Therefore, solution methodologies for Model 6.2 must use a constraint generation approach.

The constraint generation or cutting plane approach was introduced in a 1954 paper by Dantzig, Fulkerson, and Johnson [45]. These authors were using what is now known as the subtour elimination formulation of the Traveling Salesman Problem. This subtour elimination formulation has an exponential number of subtour elimination constraints, but Dantzig et al. showed that the formulation could be solved by first solving a reduced problem without any subtour constraints and then adding violated subtour constraints as needed. Dantzig et al.'s work was extremely important to the field of linear and integer programming as, according to Applegate et al. ([7], p. 91), it led to the development of both cutting plane algorithms and polyhedral combinatorics.

The usefulness of constraint generation depends on the development of a practical method for finding violated constraints. I will present two methods for generating violated constraints. The first method is simply to find the zero-forcing

closure of a solution to Model 6.2. If the closure is not the entire graph, then the set of vertices that are not in the closure forms a fort that gives a violated constraint. The second method is to use the secondary integer program in Model 6.3 to find violated forts.

For Model 6.3, define the set S to be the set of all vertices for which $s_v = 1$ in the current optimal solution of Model 6.2. Note that since the value for each s_v is taken from the current optimal solution of Model 6.2, S is constant for Model 6.3. The x_v variables indicate whether vertex v is in the fort. $N(v)$ is the open neighborhood of v (all the neighbors of v, but not v itself). The constant vector c enforces some desired property. For example, in my experiments, I set $c_v = 0.0001$ for each v to make the model find minimum size forts.

Model 6.3 Integer Program Model for Finding Forts

$$\begin{aligned}
\text{minimize} \quad & \sum_{v \in V} c_v x_v & & \\
\text{subject to:} \quad & \sum_{v \in V} x_v \geq 1 & & (1) \\
& x_w - x_v + \sum_{a \in N(w) \setminus v} x_a \geq 0 & \forall (v,w) \text{ with } v \in V,\ w \in N(v) & (2) \\
& x_v = 0 & \forall v \in zfs(S) & (3) \\
& x \in \{0,1\} & &
\end{aligned}$$

Theorem 6.5

Model 6.3 finds a minimum weight violated fort with respect to the weights given by c.

Proof: Let B be a violated fort of G. Let $x_v = 1$ for all $v \in B$ and $x_v = 0$ for all other vertices. By the definition of a fort B must contain at least one vertex; therefore, constraint (1) of the model is satisfied. Also by the definition of a fort,

any neighbor of a vertex in the fort must either be in the fort or have at least one other neighbor in the fort; therefore, constraint (2) of the model is satisfied. Finally, since B is a violated fort, there cannot exist $v \in B$ with $v \in zfs(S)$. Therefore, constraint (3) is also satisfied.

Now, let x be an solution of Model 6.3. Let B be the set of vertices of G for which $x_v = 1$. By constraint (1), B is not empty. By constraint (2), every neighbor of a vertex in B must either be in B itself or have at least two neighbors in B. By constraint (3), no vertex in B can be in $zfs(S)$. Thus, B is a violated fort of G. The result follows. ■

Model 6.3 separates violated constraints for Model 6.2. Unfortunately, solving an integer program is NP-hard. However, there is precedent in literature for using an integer programming separation method (see for example Fischetti and Lodi [73] or Avella, Boccia, and Vasilyev [11]). In our computational experiments, Model 6.3 solved quickly, and the forts found using this method are smaller and more effective at solving Model 6.2 than those found by the closure method.

Some explanation for why the forts found by Model 6.3 are more effective than those found by the closure method can be found in polyhedral theory. Since Model 6.2 is a set covering problem, the theory of Balas and Ng [16] on the set covering polytope applies. These authors gave necessary and sufficient conditions for the inequalities with right hand side of 1 (such as the fort constraints) to be facet-inducing. Their relevant theorem, restated in terms of forts, is given in Theorem 6.6.

Theorem 6.6 (Balas and Ng [16])

Given a fort B, the inequality

$$\sum_{v \in B} s_v \geq 1$$

defines a facet of the zero-forcing polytope if and only if the following two conditions hold.

1. *There does not exist a fort A with $A \subset B$.*
2. *For each $v \in V \backslash B$, there exists a $w \in B$ such that w is in all forts A with $v \in A$ and $A \subset B \cup v$.*

Condition 1 in Theorem 6.6 explains why Model 6.3 performs better than the closure method. The closure method makes no effort to minimize the size of the forts that are found. Thus, they are unlikely to satify condition 1 and be facet-inducing. On the other hand, Model 6.3 finds minimum size violated forts; therefore, the forts found cannot contain a smaller violated fort. Thus, the forts found by Model 6.3 satisfy condition 1 of Theorem 6.6. However, condition 2 is not necessarily satisfied by either method.

Although it is difficult to enforce condition 2, I will define an auxilliary integer programming model that can be used to encourage the constraints that I add to Model 6.2 to be facet-inducing. Note that if condition 2 of Theorem 6.6 is violated for a fort F, then there exist P forts $A_1, ..., A_P \subset F \cup v$ with $v \in A_i$ for $i \in \{1, ..., P\}$ but $\bigcap_{i \in \{1,...,P\}} (A_i \backslash v) = \emptyset$. Observe also that the fort constraints given by F and A_i for $i \in \{1, ..., P\}$ can be combined to give the following valid cut

$$\sum_{i \in F \cup v} x_i \geq 2$$

This valid cut is found by first summing the fort constraints corresponding to F and all the A_i. Since $\bigcap_{i \in \{1,...,P\}} A_i = \emptyset$, the coefficients of each vertex variable in the sum is at most P, but the right hand side is $P + 1$. Thus, the valid cut can be obtained by dividing through by P and taking the ceiling of each coefficient.

I make the following observation about the size of P.

Theorem 6.7

If there exist P forts $A_1, ..., A_P \subset F \cup v$ with $v \in A_i$ for $i \in \{1, ..., P\}$ but $\bigcap_{i \in \{1,...,P\}} (A_i \backslash v) = \emptyset$. Then P can be chosen to be at most $|F|$.

Proof: If $P \leq |F|$, then there is nothing to prove. On the other hand, if $P > |F|$, then for each vertex $w \in F$, choose one fort out of the P forts that does not include the vertex w. Since $\bigcap_{i \in \{1,...,P\}} (A_i \backslash v) = \emptyset$, such a fort must exist. Now, at most $|F|$ forts are chosen and the intersection of the chosen forts is empty except for v. Thus, the set of forts that are chosen has the required properties. ■

Given the above theory, I use Model 6.4 to check whether a fort generated by Model 6.3 is facet-inducing. If the generated fort is not facet-inducing, then instead of a fort constraint, I add the valid cut generated as in the previous paragraph. For Model 6.4, I use two sets of variables. The variables x_{ij} indicate whether a vertex j is chosen to be in fort i, and the variables y_i indicate whether fort i is empty.

Model 6.4 Integer Program Model for Checking if a Fort is Facet-Inducing

$$\min \quad \sum_{i \in \{1,...,|F|\}} y_i$$

$$\text{s.t.:} \quad \sum_{v \in V \backslash F} x_v = 1 \tag{1}$$

$$\sum_{v \in V \backslash F} x_{iv} = y_i \qquad \forall i \in \{1, ..., |F|\} \tag{2}$$

$$x_{iv} \leq x_v \qquad \forall i \in \{1, ..., |F|\}, \forall v \in V \backslash F \tag{3}$$

$$\sum_{i \in \{1,...,|F|\}} x_{iw} \leq \sum_{i \in \{1,...,|F|\}} y_i - 1 \quad \forall w \in F \tag{4}$$

$$x_{iw} - x_{iu} + \sum_{a \in N(w) \backslash u} x_{ia} \geq 0 \qquad u \in V, w \in N(v), \forall i \in \{1, ..., |F|\} \tag{5}$$

$$x_{iw} \leq y_i \qquad \forall i \in \{1, ..., |F|\}, \forall w \in V \tag{6}$$

$$x, y \in \{0, 1\} \tag{7}$$

Theorem 6.8

If Model 6.4 is infeasible, and F is a minimum size fort, then the fort F is facet-inducing. If Model 6.4 has an optimal solution, then the set of forts with $y_i = 1$ shows that F is not facet-inducing by property 2 of Theorem 6.6.

Proof: Suppose F is not facet-inducing. Since F is minimum size, it must satify property 1 of Theorem 6.6. Thus, F can only violate property 2 of Theorem 6.6. Therefore, there must exist P forts $A_1, ..., A_P \subset F \cup v$ with $v \in A_i$ for $i \in \{1, ..., P\}$ but $\bigcap_{i \in \{1,...,P\}} (A_i \backslash v) = \emptyset$. By Theorem 6.7, I can assume that $P \leq |F|$. Let $y_i = 1$ for $i \in \{1, ..., P\}$, and let $y_i = 0$ otherwise. Let $x_{iw} = 1$ if w is in fort A_i for $i \in \{1, ..., P\}$ and $x_{iw} = 0$ otherwise. Let $x_v = 1$ and $x_{iv} = 1$ if $i \in \{1, ..., P\}$. All variables not otherwise set are set to 0. The solution x, y defined in this manner is a feasible solution to Model 6.4. To see this fact, note that constraint (1) is satisfied because $x_w = 0$ for all $w \neq v$, and $x_v = 1$. Constraint (2) is satisfied because each fort A_i contained v. Constraint (3) is satisfied because x_{iv} is either 0 or 1 and $x_v = 1$. Constraint (4) is satisfied because $\bigcap_{i \in \{1,...,P\}} (A_i \backslash v) = \emptyset$. Constraint (5) is satisfied because each A_i was a fort, and constraint (6) is satisfied because x variables are chosen to be 1 only for the forts with y variables chosen to be 1. Thus, x, y is a feasible solution to Model 6.4. Therefore, if Model 6.4 is infeasible, then the fort F must be facet-inducing.

On the other hand, if Model 6.4 has a optimal solution, then defining the forts $A_i = \{w \in V : x_{iw} = 1\}$ gives a set of forts which shows that F does not satisfy property 2 of Theorem 6.6. ■

In addition to using Model 6.4 to determine exactly whether a fort constraint is facet-inducing, the model can also be used to determine this characteristic in a heuristic manner. For example, the model can be simplified by limiting the number

of forts that can be chosen, i.e. require at most 2 forts instead of $|F|$ forts. This simplified model is given in Model 6.5.

Model 6.5 Simplified Integer Program Model for Checking if a Fort is "Likely" to be Facet-Inducing

$$\min \quad \sum_{i \in \{1,\ldots,|F|\}} y_i$$

$$\text{s.t.:} \quad \sum_{v \in V \setminus F} x_v = 1 \tag{1}$$

$$\sum_{v \in V \setminus F} x_{iv} = 1 \qquad \forall i \in \{1,2\} \tag{2}$$

$$x_{1v} \leq x_v \qquad \forall v \in V \setminus F \tag{3}$$

$$x_{1v} + x_{2v} \leq 1 \qquad \forall v \in F \tag{4}$$

$$x_{iw} - x_{iu} + \sum_{a \in N(w) \setminus u} x_{ia} \geq 0 \quad u \in V,\ w \in N(v),\ \forall i \in \{1,2\} \tag{5}$$

$$x \in \{0,1\} \tag{7}$$

The simplified model will be easier to solve, but it will not determine exactly whether a fort constraint is facet-inducing. If a feasible solution is found for Model 6.5, then the fort constraint is not facet-inducing. However, if the simplified model is infeasible, then the constraint may or may not be facet-inducing; although, it may be more likely to be facet-inducing. In other words, Model 6.5 being infeasible is a necessary, but not sufficient, condition for the fort constraint to be facet inducing. In my computational experiments, I use the simplified model.

Although Model 6.2 is elegant and shows that the zero-forcing problem is actually a set covering problem where the sets are given by the forts of the graph, it is still not necessarily easy to solve. The model can be improved by adding variables, which I call z_v, that correspond to the vertex v and all but one of its neighboring vertices being in the forcing set. These variables will have a cost of

$|N(v)|$ where $N(v)$ is the open neighborhood of v (all the neighbors of v but not v itself). The improved model is Model 6.6.

Observe that any forcing set of a graph must contain some vertex that can force at the first iteration of the forcing rule. Therefore, any forcing set must contain a vertex, v, and all but one of v's neighbors. This property can be modeled by adding the z_v variables and requiring at least one to be positive. This property is enforced by constraint (2) of Model 6.6. Note that if a vertex, w, is in the zero-forcing closure of a the neighborhood of v and z_v is positive, then v will be forced by the corresponding solution. Therefore, x_w and z_w will never both be positive. This property is enforced by constraint (3) of Model 6.6. Finally, the fort constraints (constraint (1) of Model 6.2) must be modified to allow satisfaction by z_v variables. Despite the increased number of variables, Model 6.6 performs better in my experiments than Model 6.2.

Model 6.6 Integer Program Model with Neighborhood variables

$$\min \quad \sum_{v \in V} |N(v)| z_v + \sum_{v \in V} s_v$$

$$\text{s.t.:} \quad \sum_{v \in B} ((s_v) + \sum_{v \in zfs(N(w) \cup w)} z_w)) \geq 1 \quad \forall B \in \mathcal{B} \qquad (1)$$

$$\sum_{v \in V} z_v \geq 1 \qquad (2)$$

$$s_v + z_i \leq 1 \qquad \forall i \in V, \forall v \in zfs(N(i) \cup i) \qquad (3)$$

$$s \in \{0, 1\}$$

$$z \in \{0, 1\}$$

Given the extra variables in Model 6.6, Model 6.3 must be expanded to generate violated forts. Also, instead of minimizing the number of vertices in the fort, my preliminary experiments showed better performance from minimizing the number of vertices in the fort that are adjacent to vertices outside of the fort. Such *minimum*

border forts can be found using the integer program in Model 6.7. In this model, c_v is a penalty term to penalize vertices that are adjacent to the derived set (in our implementation all c_v values were set to 0.0001), and b_v is a binary variable that indicates whether the vertex v is adjacent to vertices outside of the fort. S is the set of all vertices, v, such that either $s_v = 1$ in the current solution of Model 6.6 or v is in the neighborhood of some vertex, w, for which $z_w = 1$ in the current solution. Constraint (3) ensures that the b_v variables correctly indicate whether the variable v is on the border of the fort.

Model 6.7 Integer Program Model for Finding Minimum Border Forts

$$\min \quad \sum_{v \in V} c_v b_v$$

$$\text{s.t.:} \quad \sum_{v \in V} x_v \geq 1 \tag{1}$$

$$x_w - x_v + \sum_{a \in N(w) \setminus v} x_a \geq 0 \qquad \forall (v,w) \text{ with } v \in V,\, w \in N(v) \tag{2}$$

$$|N(v)| x_v - |N(v)| b_v - \sum_{a \in N(v)} x_a \leq 0 \qquad \forall v \in V \tag{3}$$

$$x_v = 0 \qquad \forall v \in zfs(S) \tag{4}$$

$$x, b \in \{0,1\}$$

6.3 Computational Results for Zero-Forcing

This section presents computational results and implementation details from finding minimum zero-forcing sets using the methods previously mentioned in this chapter.

6.3.1 Implementation Details

My computational results were obtained on a Dell Precision T1650 workstation with a 3.3 GHz Intel Core i3-2120 CPU, 3.7 GB of RAM, and Red Hat Enterprise Linux

version 6.6. The code was written in C++ and compiled with g++ version 4.8. Integer programs were solved using Gurobi version 5.5.0 set to use a single thread.

Model 6.1 was simply solved in Gurobi with a MIPNODE callback to terminate the solution after 2 hours. Model 6.2 was solved in Gurobi with a POLLING callback to terminate the solution after 2 hours. A maximal set of disjoint forts was added to the formulation before solving. This maximal set is found by iteratively finding minimum size forts (using Model 6.3) that are disjoint from each other until no more such forts can be found. Other fort constraints were added to the model using a MIPSOL callback to add violated forts. Gurobi calls this callback whenever it finds a new integral incumbent solution. The callback generates minimum size violated forts by using Gurobi to solve Model 6.3. The callback can also test whether the generated fort is facet-inducing using Model 6.4.

I tested Model 6.2 both with and without testing whether forts are facet inducing. In preliminary testing, I found that the reduced Model 6.5 provided better performance than Model 6.4. This reduced version only checks if property 2 of Theorem 6.6 can be violated by 2 forts instead of up to $|F|$ forts. Although this reduced model does not guarantee the facet-inducing property, it worked better in practice because of the reduced time necessary to solve the model. Consequently, I use the reduced model for testing whether forts were facet-inducing. If a violated fort is found, the MIPSOL callback adds that fort to the formulation as a lazy constraint. If a violated fort is not found, then Gurobi terminates with an optimal solution. To enable lazy constraints, the "PreCrush" and "LazyConstraints" parameters were both set to 1. In the version of the method that checks whether forts are facet-inducing, if a generated fort is not facet-inducing, then the valid cut associated with the forts that show that the generated fort is not facet-inducing is

added instead of the generated fort.

Model 6.6 was solved similarly to Model 6.2. A POLLING callback was used to terminate after 2 hours and violated forts were added using a MIPSOL callback. I do not check whether generated forts are facet-inducing for this model. I also use Model 6.7 instead of Model 6.3 to generate violated forts that have a minimum number of vertices adjacent to vertices outside of the fort. As with Model 6.2, a maximal set of disjoint forts is added to the formulation before solving. In addition, the derived set of the closed neighborhood of each vertex gives a fort which is the set of all vertices not in the derived set. These forts were also added to the formulation before solving.

For all three models, the parameters not mentioned in the discussion above were left to their defaults in Gurobi. Some testing showed that tuning some parameters (such as the branching direction (BranchDir), aggressiveness of cut generation (Cuts), or the focus of the solver (MIPFocus)) could improve performance on some specific instances, but not in general. The branching strategy was also left to the Gurobi default. The times reported in this section are the time taken by Gurobi to optimize the relevant model. The time necessary for data input and setting up the Gurobi model is not reported.

6.3.2 Computational Tests

I test the different algorithms on three classes of random graphs: cubic, connected Watts-Strogatz [133] graphs with parameters 5 and 0.3, and connected Watts-Strogatz graphs with parameters 10 and 0.3. The parameters for the Watts-Strogatz graphs refer to the number of neighbors initially given to each vertex (A setting of 5 gives 4 neighbors to each vertex and a setting of 10 gives 10

neighbors to each vertex.) and to the probability that an edge is rewired, respectively. I used my own C++ implementation to generate random cubic graphs, and I used the connected Watts-Strogatz graph generator from the NetworkX version 1.8.1 package in Python 2.7.6. For the cubic and Watts-Strogatz graphs with parameters 5 and 0.3, I generated 5 random instances with 10, 20, ..., 100 vertices. For the Watts-Strogatz graphs with parameters 10 and 0.3, I generated 5 random instances with 20, ..., 100 vertices. I tested each algorithm until all 5 graphs of a certain size could not be solved by the algorithm within 2 hours.

The method that uses Model 6.2 without checking whether generated forts are facet-inducing can be compared with the method that does check using Model 6.5. Complete results for this comparison are given in Tables 6.13, 6.14, and 6.15, which appear on pages 134, 135, and 136, respectively. The average results for this comparison are given in Tables 6.1, 6.2, and 6.3. The results show that while checking for facet-inducing forts provides a small benefit in average running time and reduces the number of forts that must be generated, it is not effective enough to increase the size of the instances that can be solved within 2 hours. Because checking for facet-inducing forts provided no consistent benefit, subsequent results for Model 6.2 use the model without checking whether forts are facet-inducing.

Table 6.1 : Comparison of average running times for the Fort Cover IP on cubic graphs with and without checking whether forts are facet-inducing.

		Without Facets		With Facets	
$\|V\|$	Avg. Z(G)	time	forts	time	constraints
10	3.8	**0.022**	17.6	0.57	9.2
20	5.2	**0.318**	67.2	0.457	66.4
30	6.6	**1.73**	153.2	1.88	116.4
40	8.8	35.74	1834.2	**24.60**	824.6
50	9.2	274.04	4569	**200.56**	3269.6
60	11.4	**5925.79***	27082	5986.44*	27098

Note: All times are in seconds. Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported result is the average time for the instances that were successfully solved. Bold text indicates the algorithm with the best performance.

Table 6.2 : Comparison of average running times for the Fort Cover IP on Watts-Strogatz graphs with parameters (5, 0.3) with and without checking whether forts are facet-inducing.

		Without Facets		With Facets	
$\|V\|$	Avg. Z(G)	time	forts	time	constraints
10	4.4	**0.048**	22.8	0.197	21.4
20	6.2	**0.566**	102.8	1.398	100
30	7	**4.75**	245.2	5.79	245.2
40	9.4	81.06	2334.2	**73.79**	1589.6
50	10.8	2694.03*	15801	**2552.73***	14650.8
60	11.6	475.11*	1489	**438.14***	1374

Note: All times are in seconds. Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported result is the average time for the instances that were successfully solved. Bold text indicates the algorithm with the best performance.

Table 6.3 : Comparison of average running times for the Fort Cover IP on Watts-Strogatz graphs with parameters (10, 0.3) with and without checking whether forts are facet-inducing.

		Without Facets		With Facets	
$\|V\|$	Avg. Z(G)	time	forts	time	constraints
20	12	**10.49**	1133	100.48	1104
30	15.4	**1069.8**	16792.2	2492.30	16792.2

Note: All times are in seconds. Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported result is the average time for the instances that were successfully solved. Bold text indicates the algorithm with the best performance.

Table 6.4 gives the size of instances for which each method failed on at least one instance. These results show that the Fort Cover IP (Model 6.2) and the Extended Cover IP (Model 6.6) perform similarly and they are both much better than the Infection IP (Model 6.1). However, Wavefront performs better than any of the integer programs and is also less sensitive to the density of the graph.

Table 6.4 : Size of graphs where methods start to fail.

Graph Type	Wavefront	Infection IP	Fort Cover IP	Extended Cover IP
Cubic	**80**	30	60	60
WS (5, 0.3)	**80**	30	50	60
WS (10, 0.3)	**80**	20	40	40

Note: Bold text indicates the algorithm with the best performance. Wavefront can handle the largest graphs and also is less sensitive to the density of the graphs.

Table 6.5 gives the average results for cubic graphs over each graph size for each algorithm. These results again show that the Wavefront algorithm performs best, followed by the Extended Cover IP. While the size of the instances that Fort Cover IP and the Extended Cover IP can handle is similar, the Extended Cover IP solves the instances faster on average. The complete results for cubic graphs are given in Table 6.10 on page 131.

Tables 6.6 and 6.7 give the average results for the Watts-Strogatz graphs with parameters (5,0.3) and (10,0.3) respectively. These results are similar to those for cubic graphs; however, they indicate that the Wavefront algorithm is less sensitive to changes in the graph structure than the integer programs. The results also indicate that both integer programs are sensitive to the degree of vertices in the graph, but the Extended Cover IP (Model 6.6) seems to be more sensitive than the Fort Cover IP (Model 6.2). The Fort Cover IP actually beats the Extended Cover IP for the instances with parameters 10 and 0.3. The complete results for the Watts-Strogatz graphs are in Tables 6.11 and 6.12 on pages 132 and 133, respectively.

Although the Wavefront algorithm performs best on the cubic and Watts-Strogatz graphs, it does not perform well on stars. Table 6.8 compares the performance of Wavefront and the Fort Cover IP on stars of up to 101 vertices. While the Fort Cover IP is able to solve all of these stars quickly, the Wavefront algorithm fails for stars with only 31 vertices.

Table 6.5 : Average running times for zero-forcing algorithms on random cubic graphs.

$\|V\|$	Avg. Z(G)	Wavefront	Infection	Fort Cover	Extended Cover
10	3.8	**0.0013**	0.0470	0.022	0.024
20	5.2	**0.017**	77.43	0.32	0.11
30	6.6	**0.18**	206.20*	1.73	0.64
40	8.8	**2.79**	T	35.74	7.18
50	9.2	**9.68**	T	274.04	40.33
60	11.4	**227.02**	T	5925.79*	1813.21*
70	12	**525.46**	T	T	T
80	12	**681.13***	T	T	T

Note: All times are in seconds. Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported result is the average time for the instances that were successfully solved. T indicates that none of the 5 instances were solved within 2 hours. Bold text indicates the algorithm with the best performance.

Table 6.6 : Average running times for zero-forcing algorithms on random connected Watts-Strogatz graphs with parameters (5, 0.3).

$\|V\|$	Avg. Z(G)	Wavefront	Infection	Fort Cover	Extended Cover
10	4.4	**0.0013**	0.70	0.048	0.047
20	6.2	**0.018**	152.47	0.57	0.55
30	7	**0.072**	5320.23*	4.75	3.58
40	9.4	**1.32**	T	81.06	47.58
50	10.8	**10.47**	T	2694.03	2234.27
60	11.6	**69.30**	T	475.11*	2387.36*
70	14	**678.89**	T	T	T
80	14.5	**1306.57***	T	T	T

Note: All times are in seconds. Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported result is the average time for the instances that were successfully solved. T indicates that none of the 5 instances were solved within 2 hours. Bold text indicates the algorithm with the best performance.

Table 6.7 : Average running times for zero-forcing algorithms on random connected Watts-Strogatz graphs with parameters (10, 0.3).

$\|V\|$	Avg. Z(G)	Wavefront	Infection	Fort Cover	Extended Cover
20	12	**0.010**	T	10.47	11.52
30	15.4	**0.11**	T	1069.80	1370.70
40	18	**0.93**	T	T	T
50	21.8	**9.41**	T	T	T
60	24.6	**65.51**	T	T	T
70	27.4	**416.65**	T	T	T
80	30	**2192.86***	T	T	T

Note: All times are in seconds. Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported result is the average time for the instances that were successfully solved. T indicates that none of the 5 instances were solved within 2 hours. Bold text indicates the algorithm with the best performance.

Table 6.8 : Comparison of running times for Wavefront and the Fort Cover IP on stars.

$\|V\|$	Z(G)	Wavefront	Fort Cover
11	9	0.41	**0.03**
21	19	3087.20	**0.15**
31	29	K	**0.46**
41	39	K	**0.94**
51	49	K	**2.19**
61	59	K	**4.10**
71	69	K	**7.44**
81	79	K	**12.72**
91	89	K	**19.86**
101	99	K	**31.67**

Note: All times are in seconds. K indicates that the algorithm ran out of memory. Bold text indicates the algorithm with the best performance.

Some additional insights can be gained by looking at the time taken to generate forts in the Fort Cover and Extended Cover integer programs. Table 6.9 gives the average percentage of the time spent by these two methods that was spent generating violated forts for all three types of graphs. These results show that the Extended Cover IP spends a lower percentage of its time generating forts than the Fort Cover IP. Both integer programs spend the majority of their time generating forts. However, this percentage decreases when the instances reach a size where the methods start to fail. This percentage then increases again as the size of failure is surpassed. This behavior indicates that at the size of failure, Gurobi was not able to generate as many incumbent solutions and spent more time trying to prove optimality. This behavior makes sense because as the set of fort constriants in the model approaches a set of forts necessary to solve the zero-forcing problem, then generating additional feasible solutions of lower cost will become more difficult, and the solver will spend more time proving optimality.

Table 6.9 : Average percentage of time spent generating forts.

	Cubic		WS (5,0.3)		WS (10,0.3)	
$\|V\|$	Fort Cover	Ext. Cover	Fort Cover	Ext. Cover	Fort Cover	Ext. Cover
10	75.9	72.6	88.8	85.9	N/A	N/A
20	92.6	56.5	93.4	91.1	91.4	88.9
30	89.9	74.9	89.6	86.2	52.4	39.9
40	77.9	55.4	71.9	67.3	36.1*	32.6*
50	73.1	58.9	46.0*	48.1		
60	20.8*	23.8*	93.8*	86.6*		
70	57.0*	71.1*	82.7*	93.1*		

Note: Asterisks indicate that not all instances of the specified size were solved within 2 hours. In these cases, the reported percentage is the percentage of the 2 hours that was used to generate forts.

Table 6.10 : Running times for zero-forcing algorithms on random cubic graphs.

		Wavefront	Infection	Fort Cover			Extended Cover		
Graph	Z(G)	time	time	time	forts	fort time	time	forts	fort time
cubic10_1	4	**0.001**	0.070	0.025	19	0.017	0.030	20	0.023
cubic10_2	4	**0.001**	0.055	0.026	19	0.019	0.032	21	0.023
cubic10_3	4	**0.002**	0.092	0.033	23	0.029	0.038	22	0.024
cubic10_4	4	**0.002**	0.067	0.012	17	0.007	0.014	18	0.010
cubic10_5	3	**0.0005**	0.065	0.013	10	0.012	0.005	8	0.004
cubic20_1	5	**0.015**	49.38	0.335	58	0.318	0.097	28	0.048
cubic20_2	5	**0.011**	31.61	0.180	46	0.167	0.041	27	0.029
cubic20_3	6	**0.039**	219.87	0.453	119	0.401	0.235	63	0.157
cubic20_4	5	**0.011**	49.81	0.273	51	0.254	0.099	31	0.029
cubic20_5	5	**0.009**	36.47	0.349	62	0.328	0.071	30	0.047
cubic30_1	8	0.504	539.58	1.70	221	1.51	**0.38**	73	0.34
cubic30_2	6	**0.061**	T	1.40	102	1.33	0.19	43	0.15
cubic30_3	7	**0.206**	T	4.09	317	2.96	2.40	123	0.96
cubic30_4	6	**0.057**	29.10	0.64	52	0.62	0.11	34	0.10
cubic30_5	6	**0.058**	49.91	0.82	74	0.79	0.12	36	0.09
cubic40_1	9	**2.84**	T	20.51	910	14.99	5.66	212	3.26
cubic40_2	9	**3.24**	T	31.93	2064	27.16	6.66	239	3.87
cubic40_3	9	**3.68**	T	78.56	3988	56.58	14.71	425	6.87
cubic40_4	9	**3.10**	T	38.43	1830	28.41	6.32	262	3.75
cubic40_5	8	**1.09**	T	9.27	379	7.93	2.55	106	1.41
cubic50_1	9	**6.49**		64.37	1283	54.65	14.50	256	8.19
cubic50_2	9	**5.39**		134.18	2090	106.40	16.83	271	8.96
cubic50_3	8	**1.92**		25.91	402	23.40	12.54	174	9.83
cubic50_4	10	**16.7**		377.05	8864	260.38	41.23	603	17.12
cubic50_5	10	**17.88**		768.71	10206	322.05	116.53	2230	75.48
cubic60_1	11	**126.99**		T	20819	1307.51	T	11587	1083.08
cubic60_2	13	**686.62**		T	22586	793.37	T	14200	725.40
cubic60_3	11	**101.59**		T	23412	1415.18	T	21254	1819.79
cubic60_4	11	**120.52**		T	31360	1894.49	T	14359	1264.73
cubic60_5	11	**99.4**		5925.79	27082	1701.73	1813.21	13061	921.07
cubic70_1	12	**366.54**		T	22052	3317.08	T	16028	3149.15
cubic70_2	13	**1362.46**		T	24073	2472.74	T	28902	5194.53
cubic70_3	11	**138.38**		T	27041	5645.79	T	13911	6417.39
cubic70_4	12	**372.85**		T	30200	4423.89	T	24618	5984.35
cubic70_5	12	**387.08**		T	37628	4674.44	T	21687	4868.10
cubic80_1		K							
cubic80_2		K							
cubic80_3		K							
cubic80_4	12	**656.72**							
cubic80_5	12	**705.54**							

Note: All times are in seconds. The time columns give the total time required. The fort time columns give the time used to generate forts. The forts columns give the number of forts generated. T indicates the algorithm did not find a solution within 2 hours. K indicates that the algorithm ran out of memory. Bold text indicates the algorithm with the best performance.

Table 6.11 : Running times for zero-forcing algorithms on random connected Watts-Strogatz graphs with parameters (5, 0.3).

		Wavefront	Infection	Fort Cover			Extended Cover		
Graph	Z(G)	time	time	time	forts	fort time	time	forts	fort time
WS10_5_0.3_1	4	**0.0003**	0.52	0.034	15	0.031	0.038	19	0.032
WS10_5_0.3_2	4	**0.0005**	0.44	0.031	15	0.027	0.033	19	0.029
WS10_5_0.3_3	4	**0.0004**	0.62	0.052	17	0.047	0.042	19	0.036
WS10_5_0.3_4	5	**0.0028**	1.02	0.054	32	0.047	0.050	37	0.042
WS10_5_0.3_5	5	**0.0027**	0.92	0.067	35	0.059	0.072	41	0.063
WS20_5_0.3_1	6	**0.014**	291.61	0.55	94	0.52	0.56	89	0.52
WS20_5_0.3_2	6	**0.0098**	241.28	0.54	78	0.52	0.51	66	0.47
WS20_5_0.3_3	6	**0.0135**	81.46	0.55	84	0.52	0.51	85	0.47
WS20_5_0.3_4	7	**0.035**	105.63	0.61	171	0.53	0.65	151	0.56
WS20_5_0.3_5	6	**0.019**	42.38	0.58	87	0.55	0.50	79	0.46
WS30_5_0.3_1	6	**0.020**	T	2.65	102	2.58	1.25	84	1.19
WS30_5_0.3_2	8	**0.18**	T	9.21	559	7.38	7.29	335	5.46
WS30_5_0.3_3	7	**0.052**	T	4.29	204	3.74	3.53	155	2.78
WS30_5_0.3_4	7	**0.053**	5320.23	3.70	184	3.46	2.97	143	2.69
WS30_5_0.3_5	7	**0.056**	T	3.91	177	3.52	2.84	136	2.60
WS40_5_0.3_1	9	**0.88**	T	50.81	1379	39.22	26.14	581	18.83
WS40_5_0.3_2	9	**0.67**	T	75.17	1551	47.92	53.28	809	32.01
WS40_5_0.3_3	10	**2.33**	T	82.37	2678	60.84	68.57	1955	52.85
WS40_5_0.3_4	10	**1.89**	T	118.95	4591	88.52	47.16	1364	32.58
WS40_5_0.3_5	9	**0.83**	T	77.98	1472	54.65	42.73	606	24.95
WS50_5_0.3_1	11	**11.32**		2888.01	16692	1024.05	1016.52	7237	543.21
WS50_5_0.3_2	11	**11.58**		3298.80	19076	1181.12	1861.02	13775	1019.08
WS50_5_0.3_3	10	**4.01**		742.90	6639	563.88	246.55	1624	150.09
WS50_5_0.3_4	11	**15.11**		3846.42	20797	1369.44	2245.79	14450	1141.41
WS50_5_0.3_5	11	**10.34**		T	46162	3382.67	5801.49	12730	1204.41
WS60_5_0.3_1	11	**17.14**		T	23938	5331.91	3527.24	7140	2874.11
WS60_5_0.3_2	13	**172.46**		T	30016	3059.41	T	32814	4978.93
WS60_5_0.3_3	10	**5.46**		475.11	1489	445.70	196.22	458	186.67
WS60_5_0.3_4	13	**134.9**		T	27404	3547.67	T	32417	6576.26
WS60_5_0.3_5	11	**16.54**		T	26113	5378.09	3438.63	9926	2861.53
WS70_5_0.3_1	14	**680.95**		T	19577	6487.26	T	14466	6932.66
WS70_5_0.3_2	13	**109.31**		T	15068	6518.36	T	8742	6831.20
WS70_5_0.3_3	14	**406.7**		T	19410	6608.86	T	10355	6994.96
WS70_5_0.3_4	14	**596.85**		T	23392	5742.50	T	20093	6974.45
WS70_5_0.3_5	15	**1600.63**		T	21508	4430.44	T	17387	5776.49
WS80_5_0.3_1		K							
WS80_5_0.3_2		K							
WS80_5_0.3_3	14	**859.39**							
WS80_5_0.3_4		K							
WS80_5_0.3_5	15	**1753.74**							

Note: All times are in seconds. The time columns give the total time required. The fort time columns give the time used to generate forts. The forts columns give the number of forts generated. T indicates the algorithm did not find a solution within 2 hours. K indicates that the algorithm ran out of memory. Bold text indicates the algorithm with the best performance.

Table 6.12 : Running times for zero-forcing algorithms on random connected Watts-Strogatz graphs with parameters (10, 0.3).

		Wavefront	Infection	Fort Cover			Extended Cover		
Graph	Z(G)	time	time	time	forts	fort time	time	forts	fort time
WS20_10_0.3_1	12	**0.012**	T	12.15	1137	11.29	12.43	1047	11.29
WS20_10_0.3_2	12	**0.0071**	T	10.52	1170	9.51	12.82	1332	11.49
WS20_10_0.3_3	12	**0.0080**	T	10.42	1077	9.58	12.25	1250	11.10
WS20_10_0.3_4	12	**0.011**	T	8.92	1009	7.98	9.99	971	8.65
WS20_10_0.3_5	12	**0.012**	T	10.42	1272	9.64	10.12	1069	8.79
WS30_10_0.3_1	15	**0.086**		785.53	14111	449.48	948.81	12733	435.29
WS30_10_0.3_2	16	**0.16**		1361.35	21037	560.35	1797.93	16548	448.42
WS30_10_0.3_3	14	**0.039**		480.65	10763	418.68	489.51	10088	391.99
WS30_10_0.3_4	16	**0.14**		1464.43	21577	599.60	1787.75	15418	428.27
WS30_10_0.3_5	16	**0.13**		1257.04	16473	445.57	1829.52	15337	450.75
WS40_10_0.3_1	19	**1.80**		T	49527	2932.56	T	31067	2194.36
WS40_10_0.3_2	17	**0.53**		T	27590	1929.74	T	20556	1666.13
WS40_10_0.3_3	18	**0.75**		T	39648	3247.84	T	30816	2636.68
WS40_10_0.3_4	18	**0.72**		T	30868	2429.47	T	25028	2301.58
WS40_10_0.3_5	18	**0.84**		T	31577	2453.37	T	30280	2897.96
WS50_10_0.3_1	22	**10.72**							
WS50_10_0.3_2	22	**9.87**							
WS50_10_0.3_3	22	**9.54**							
WS50_10_0.3_4	21	**5.32**							
WS50_10_0.3_5	22	**11.6**							
WS60_10_0.3_1	23	**24.61**							
WS60_10_0.3_2	25	**66.62**							
WS60_10_0.3_3	25	**70.78**							
WS60_10_0.3_4	25	**81.22**							
WS60_10_0.3_5	25	**84.33**							
WS70_10_0.3_1	26	**137.08**							
WS70_10_0.3_2	29	**854.47**							
WS70_10_0.3_3	28	**554.35**							
WS70_10_0.3_4	27	**295.78**							
WS70_10_0.3_5	27	**241.59**							
WS80_10_0.3_1		K							
WS80_10_0.3_2		K							
WS80_10_0.3_3		K							
WS80_10_0.3_4		K							
WS80_10_0.3_5	30	**2192.86**							

Note: All times are in seconds. The time columns give the total time required. The fort time columns give the time used to generate forts. The forts columns give the number of forts generated. T indicates the algorithm did not find a solution within 2 hours. K indicates that the algorithm ran out of memory. Bold text indicates the algorithm with the best performance.

Table 6.13 : Comparison of Fort Cover IP with and without checking for facets on cubic graphs.

Graphs	Z(G)	Without Facets			With Facets		
		time	forts	fort time	time	constraints	fort time
cubic10_1	4	**0.025**	19	0.017	0.053	5	0.049
cubic10_2	4	**0.026**	19	0.019	0.047	5	0.044
cubic10_3	4	**0.033**	23	0.029	0.145	19	0.140
cubic10_4	4	**0.012**	17	0.007	0.017	7	0.015
cubic10_5	3	**0.013**	10	0.012	0.021	10	0.020
cubic20_1	5	**0.335**	58	0.318	0.434	57	0.417
cubic20_2	5	**0.180**	46	0.167	0.288	43	0.275
cubic20_3	6	**0.453**	119	0.401	0.741	119	0.687
cubic20_4	5	**0.273**	51	0.254	0.356	51	0.337
cubic20_5	5	**0.349**	62	0.328	0.468	62	0.447
cubic30_1	8	1.70	221	1.51	**1.43**	70	1.36
cubic30_2	6	**1.40**	102	1.33	1.62	102	1.56
cubic30_3	7	**4.09**	317	2.96	4.86	317	3.71
cubic30_4	6	0.64	52	0.62	**0.50**	19	0.49
cubic30_5	6	**0.82**	74	0.79	0.98	74	0.95
cubic40_1	9	20.51	910	14.99	**16.96**	586	12.27
cubic40_2	9	31.93	2064	27.16	**11.15**	216	8.98
cubic40_3	9	78.56	3988	56.58	**48.34**	1522	36.63
cubic40_4	9	**38.43**	1830	28.41	38.58	1561	29.68
cubic40_5	8	9.27	379	7.93	**7.97**	238	7.05
cubic50_1	9	64.37	1283	54.65	**34.90**	524	28.34
cubic50_2	9	**134.18**	2090	106.40	147.01	2335	119.31
cubic50_3	8	25.91	402	23.40	**20.90**	290	18.45
cubic50_4	10	**377.05**	8864	260.38	378.79	7077	242.45
cubic50_5	10	768.71	10206	322.05	**421.21**	6122	238.22
cubic60_1	11	T	20819	1307.51	T	20782	1380.10
cubic60_2	13	T	22586	793.37	T	9792	632.30
cubic60_3	11	T	23412	1415.18	T	23373	1494.61
cubic60_4	11	T	31360	1894.49	T	32694	2095.95
cubic60_5	11	**5925.79**	27082	1701.73	5986.44	27098	2237.68
cubic70_1	12	T	22052	3317.08	T	15878	2753.51
cubic70_2	13	T	24073	2472.74	T	20535	2964.96
cubic70_3	11	T	27041	5645.79	T	27002	5734.30
cubic70_4	12	T	30200	4423.89	T	15709	2915.01
cubic70_5	12	T	37628	4674.44	T	30678	4053.19

Note: All times are in seconds. The time columns give the total time required. The fort time columns give the time used to generate forts or constraints. The forts or constraints columns give the number of forts or constraints generated. T indicates the algorithm did not find a solution within 2 hours. Bold text indicates the algorithm with the best performance.

Table 6.14 : Comparison of Fort Cover IP with and without checking for facets on random connected Watts-Strogatz graphs with parameters (5, 0.3).

Graphs	Z(G)	Without Facets			With Facets		
		time	forts	fort time	time	constraints	fort time
WS10_5_0.3_1	4	**0.034**	15	0.031	0.064	15	0.060
WS10_5_0.3_2	4	**0.031**	15	0.027	0.147	15	0.143
WS10_5_0.3_3	4	**0.052**	17	0.047	0.143	17	0.138
WS10_5_0.3_4	5	**0.054**	32	0.047	0.284	26	0.276
WS10_5_0.3_5	5	**0.067**	35	0.059	0.345	34	0.337
WS20_5_0.3_1	6	**0.55**	94	0.52	1.21	94	1.18
WS20_5_0.3_2	6	**0.54**	78	0.52	0.972	64	0.947
WS20_5_0.3_3	6	**0.55**	84	0.52	1.09	84	1.06
WS20_5_0.3_4	7	**0.61**	171	0.53	2.56	172	2.48
WS20_5_0.3_5	6	**0.58**	87	0.55	1.16	86	1.13
WS30_5_0.3_1	6	**2.65**	102	2.58	3.00	102	2.93
WS30_5_0.3_2	8	**9.21**	559	7.38	11.87	559	10.02
WS30_5_0.3_3	7	**4.29**	204	3.74	5.01	204	4.46
WS30_5_0.3_4	7	**3.70**	184	3.46	4.46	184	4.21
WS30_5_0.3_5	7	**3.91**	177	3.52	4.61	177	4.22
WS40_5_0.3_1	9	**50.81**	1379	39.22	58.62	1384	47.38
WS40_5_0.3_2	9	**75.17**	1551	47.92	81.23	1551	54.15
WS40_5_0.3_3	10	82.37	2678	60.84	**59.26**	1228	46.01
WS40_5_0.3_4	10	118.95	4591	88.52	**85.81**	2313	59.65
WS40_5_0.3_5	9	**77.98**	1472	54.65	84.02	1472	60.95
WS50_5_0.3_1	11	2888.01	16692	1024.05	**2361.62**	14859	1004.96
WS50_5_0.3_2	11	3298.80	19076	1181.12	**3196.08**	15868	1056.45
WS50_5_0.3_3	10	**742.90**	6639	563.88	771.43	6639	592.43
WS50_5_0.3_4	11	**3846.42**	20797	1369.44	3881.79	21237	1494.53
WS50_5_0.3_5	11	T	46162	3382.67	T	46147	3599.26
WS60_5_0.3_1	11	T	23938	5331.91	T	23536	5380.92
WS60_5_0.3_2	13	T	30016	3059.41	T	30422	3472.54
WS60_5_0.3_3	10	475.11	1489	445.70	**438.14**	1374	412.96
WS60_5_0.3_4	13	T	27404	3547.67	T	27311	3688.85
WS60_5_0.3_5	11	T	26113	5378.09	T	25752	5420.59
WS70_5_0.3_1	14	T	19577	6487.26	T	19442	6559.93
WS70_5_0.3_2	13	T	15068	6518.36	T	13967	6586.92
WS70_5_0.3_3	14	T	19410	6608.86	T	19280	6673.42
WS70_5_0.3_4	14	T	23392	5742.50	T	22614	5811.77
WS70_5_0.3_5	15	T	21508	4430.44	T	21404	4527.84

Note: All times are in seconds. The time columns give the total time required. The fort time columns give the time used to generate forts or constraints. The forts or constraints columns give the number of forts or constraints generated. T indicates the algorithm did not find a solution within 2 hours. K indicates that the algorithm ran out of memory. Bold text indicates the algorithm with the best performance.

Table 6.15 : Comparison of Fort Cover IP with and without checking for facets on random connected Watts-Strogatz graphs with parameters (10, 0.3).

Graphs	Z(G)	Without Facets			With Facets		
		time	forts	fort time	time	constraints	fort time
WS20_10_0.3_1	12	**12.15**	1137	11.29	108.66	1180	107.71
WS20_10_0.3_2	12	**10.52**	1170	9.51	99.51	1087	98.49
WS20_10_0.3_3	12	**10.42**	1077	9.58	107.76	1210	106.75
WS20_10_0.3_4	12	**8.92**	1009	7.98	90.80	960	89.89
WS20_10_0.3_5	12	**10.42**	1272	9.64	95.69	1083	94.74
WS30_10_0.3_1	15	**785.53**	14111	449.48	1880.72	14111	1545.74
WS30_10_0.3_2	16	**1361.35**	21037	560.35	3134.54	21037	2335.21
WS30_10_0.3_3	14	**480.65**	10763	418.68	1242.58	10763	1180.34
WS30_10_0.3_4	16	**1464.43**	21577	599.60	3283.18	21577	2418.98
WS30_10_0.3_5	16	**1257.04**	16473	445.57	2920.50	16473	2106.39
WS40_10_0.3_1	19	T	49527	2932.56	T	35494	5085.25
WS40_10_0.3_2	17	T	27590	1929.74	T	24238	3858.45
WS40_10_0.3_3	18	T	39648	3247.84	T	35353	5119.25
WS40_10_0.3_4	18	T	30868	2429.47	T	27077	4287.48
WS40_10_0.3_5	18	T	31577	2453.37	T	29593	4060.35

Note: All times are in seconds. The time columns give the total time required. The fort time columns give the time used to generate forts or constraints. The forts or constraints columns give the number of forts or constraints generated. T indicates the algorithm did not find a solution within 2 hours. Bold text indicates the algorithm with the best performance.

6.4 Connected Zero-Forcing

It is common for many graph problems to have a connected version. For example, the dominating set and power dominating set problems also have interesting connected variants. Likewise, one can consider a connected version of the zero-forcing problem. In the *connected* zero-forcing problem, the vertices forming the zero-forcing set must induce a connected subgraph. The connected version of zero-forcing has been studied by Brimkov and Davila [26], who gave formulas for the connected forcing number of certain classes of graphs; Davila, Henning, Magnant, and Pepper [48], who gave bounds on the connected forcing number using certain graph invariants; and by Brimkov [25], who showed that the zero-forcing problem is still NP-hard in the connected version. However, to my knowledge, there have not been any efforts to develop computational tools for finding minimum connected zero-forcing sets. In this section, I develop computational methods for connected zero-forcing.

As mentioned in the previous section, the only computational method previously known for the zero-forcing problem is the Wavefront algorithm described in Algorithm 6.1. However, Wavefront fails for connected forcing because of how the algorithm constructs zero-forcing sets. For a graph, G, Wavefront finds and stores optimal forcing sets for certain subgraphs of G. Wavefront then builds the optimal forcing sets of the larger subgraphs by adding neighborhoods of vertices that have an unforced vertex in them. However, the zero-forcing sets produced in this manner are not necessarily connected.

While Wavefront could potentially be modified to create connected forcing sets, such a modification would eliminate the computational advantages of the algorithm. Wavefront has good performance because it only stores the optimal forcing set for a

certain subgraph; however, this optimal forcing set for the subgraph may not be connected to other vertices that must be added to the forcing set to force all of G. Thus, to be useful for finding connected forcing sets, Wavefront would need to store more than just the optimal forcing set for each subgraph, and Wavefront's performance would suffer as a result. For these reasons, Wavefront will not be a viable method for solving the connected zero-forcing problem without significant alterations that are beyond the scope of this thesis.

6.4.1 Branch and Bound Algorithm

The connected zero-forcing problem can, of course, be solved by a brute force approach that simply generates subsets of vertices with a certain size, checks if they are connected, and then checks if they form a zero-forcing set. However, I give next a branch and bound style algorithm that generates only connected subgraphs, checks if they form zero-forcing sets, and prunes the search tree based on the best zero-forcing set found. The central part of this method is the generation of connected subgraphs, since all connected subgraphs must be generated and I do not test for connectivity.

Avis and Fukuda [14] showed that a reverse search algorithm will generate the connected induced subgraphs of a graph. The algorithm that I present here is an implementation of reverse search. The algorithm is based on the idea that the connected induced subgraphs of a given graph can be generated by starting from a subset containing each single vertex of the graph and recursively adding a neighbor of that subset to the subset of vertices. In essence, the connected subgraphs of the graph are given by the leaves of a tree defined by the choice of whether or not a certain neighbor of the current subset is in the connected subgraph. Given a subset

of vertices, the choice of whether or not a certain neighbor is in the subgraph gives two branches of a subtree descending from a vertex representing the current subset.

This tree can be searched, using for example depth-first search, to generate all connected subgraphs. In addition, each subtree will only include subsets of vertices that are larger than the subset represented by the root of the subtree. Thus, once I find a connected zero-forcing set of a certain size, I can prune all branches of the tree that lead to subsets of equal or greater size. Algorithm 6.2 gives the psuedocode for my algorithm.

6.4.2 Integer Programming Methods

While the Wavefront algorithm is difficult to adapt to connected zero-forcing, the integer programming models introduced in the previous section can be adapted to the problem simply by adding constraints to enforce connectivity on the chosen zero-forcing set. I focus on adding connectivity constraints to Model 6.2 because it is the best performing model from the previous section that allows us to ensure connectivity. Drawing from the literature on connected dominating sets and connected power dominating sets, there are multiple ways of modeling connectivity. Fan and Watson [66] compared four methods for enforcing connectivity in integer programs: Miller-Tucker-Zemlin (MTZ) constraints, Martin constraints, single-commodity flow constraints, and multi-commodity flow constraints. They found that the MTZ constraints provided the best computational performance for both the connected dominating set and connected power dominating set problems. Another method of enforcing connectivity is to add a,b-separation cutting planes when needed to cut off disconnected solutions. This method has been used by Buchanan, Sang Sung, Butenko, and Pasiliao [29] for connected dominating sets; by

Algorithm 6.2: Branch and bound connected zero-forcing algorithm

Data: A graph G; three sets of vertices Not, S, $N(S)$; and a constant L
Result: A minimum connected zero-forcing set of G
if *Not, S, and $N(S)$ are not initialized* **then**
  $Not = V$, $S = \emptyset$, $N(S) = \emptyset$, $L = |V|$;
end
if *S is empty* **then**
  $C = Not$;
end
else
  $C = Not \cap N(S)$;
end
if *C is empty* **then**
  if *S is a zero-forcing set* **then**
    $L = |S|$;
    return S;
  end
  else
    return;
  end
end
else
  Choose any $v \in C$;
  Algorithm 6.2($Not - v$, S, $N(S)$, L);
  if $|S| < L - 1$ **then**
    Algorithm 6.2($Not - v$, $S \cup v$, $N(S) \cup neighbors(v)$, L);
  end
end

Fishetti et al. [72] for Steiner trees; and by Carvajal et al [38] for forest planning problems. Wang, Buchanan, and Butenko [131] studied conditions that cause such inequalities to induce facets of the connected subgraph polytope.

In this section, I compare the use of MTZ constraints and a,b-separation inequalities to enforce connectivity for the connected zero-forcing problem. I also compare to a brute force method that generates all connected subsets of vertices and tests to see if they are zero-forcing, and to the branch and bound method in Algorithm 6.2.

MTZ constraints were originally introduced by Miller, Tucker, and Zemlin [98] to study the Traveling Salesman Problem. The basic idea of MTZ constraints is to enforce the existence of a directed spanning tree in the subgraph induced by the chosen vertices. For my implementation, I follow Fan and Watson's [66] explanation of the method introduced by Quintão, da Cunha, Mateus, and Lucena [107]. In this implementation, two new vertices, α and β are added to the graph, and a set E_{new} of edges is also added. E_{new} contains a directed edge from each of the two new vertices to all the original vertices (I continue to denote the set of original vertices by V). E_{new} also contains a directed edge from α to β. The idea behind this alteration of the graph is that the vertices that are not chosen to be in the forcing set will have a positive edge variable coming into them from α, while β will have a positive edge variable going to the root of the directed spanning tree of the chosen connected zero-forcing set. Model 6.2 combines with the MTZ constraints to form Model 6.8.

Model 6.8 Integer Program Model of the Connected Zero-Forcing Problem using MTZ Constraints

$$\text{Min.} \quad \sum_{v \in V} s_v$$

$$\begin{aligned}
\text{S.t.:} \quad & \sum_{v \in B} s_v \geq 1 && \forall B \in \mathcal{B} && (1)\\
& \sum_{v \in V} y_{\beta,v} = 1 && && (2)\\
& \sum_{i:(i,v) \in E} y_{i,v} = 1 && \forall v \in V && (3)\\
& y_{\alpha,v} + y_{v,i} \leq 1 && \forall (v,i) \in E && (4)\\
& (n+1)y_{i,v} + u_i - u_v + (n-1)y_{v,i} \leq n && \forall (v,i) \in E && (5)\\
& (n+1)y_{i,v} + u_i - u_v \leq n && \forall (v,i) \in E_{new} && (6)\\
& x_v = 1 - y_{\alpha,v} && \forall v \in V && (7)\\
& y_{\alpha,\beta} = 1 && && (8)\\
& u_\alpha = 0 && && (9)\\
& 1 \leq u_v \leq n+1 && \forall v \in V \cup \{\beta\} && (10)\\
& s \in \{0,1\} && && (11)\\
& y \in \{0,1\} && && (12)\\
& u \in \mathbb{Z} && && (13)
\end{aligned}$$

In Model 6.8, the constraints (1) and (11) are the original zero-forcing constraints from Model 6.2. The rest of the constraints are the MTZ constraints. Constraint (2) ensures that there is an edge chosen from β to some vertex that will be the root of the directed spanning tree of the zero-forcing set. Constraint (3) ensures that each vertex has an incoming edge. Constraint (4) ensures that vertices connected to α cannot be used to connect to any other vertices. Constraints (5) and (6) ensures that there are no cycles in the chosen edges. Constraint (7) ensures that vertices chosen to be in the forcing set must be in the spanning tree instead of connected to α. The rest of the constraints are just bounds.

A solution of Model 6.8 is an optimal connected zero-forcing set; however, the MTZ constraints require that the number of variables in the model is more than triple that of Model 6.2. Furthermore, some of the variables in Model 6.8 are integer instead of binary. A second method for finding connected forcing sets does not require additional variables in the model. Instead of additional variables, valid inequalities can be added that cut off disconnected solutions. As previously mentioned, this method has been used for other problems in the literature (see for example [29], [72], and [38]).

The valid inequalities that I will add to Model 6.2 are known as a,b-separation inequalities. The idea behind these constraints is that if a set C of vertices is a vertex cut separating two vertices a and b in a graph and both a and b are chosen to be in the zero-forcing set, then some vertex from C must also be chosen to be in the zero-forcing set. Model 6.9 gives the complete integer programming model for connected zero-forcing using a,b-separation inequalities. Constraints (1) and (3) are the zero-forcing constraints from Model 6.2, and the constraints (2) are the a,b-separation inequalities.

Model 6.9 Integer Program Model of the Connected Zero-Forcing Problem using a,b-separators

$$\text{Min.} \quad \sum_{v \in V} s_v$$

$$\text{S.t.:} \quad \sum_{v \in B} s_v \geq 1 \qquad \forall B \in \mathcal{B} \qquad (1)$$

$$s_a + s_b - \sum_{v \in C} s_v \leq 1 \qquad \forall \text{ pairs } a, b \in V, C \text{ an a,b-separator} \qquad (2)$$

$$s \in \{0, 1\} \qquad (3)$$

The a,b-separation inequalities can be separated efficiently using the observation

that if the chosen zero-forcing set Z is not connected, then the set $C = V \backslash Z$ must be a vertex cut separating at least two vertices $a \in Z$ and $b \in Z$. However, as was pointed out by Buchanan et al. [29], the resulting vertex cuts are likely larger than necessary. Since the decision variable for each vertex in C appears in these constraints, the constraints are stronger when the size of the vertex cut, S, is minimized. Therefore, Buchanan et al. [29] gave an algorithm for deleting vertices from a vertex cut of G until it became inclusion minimal. For the dominating set problem, a valid cutting plane can be obtained from a vertex cut; however, a zero-forcing set does not have to be dominating. Therefore, I also require that the vertex cut must be an a,b-separator for $a, b \in Z$. In Algorithm 6.3, I give a modified version of Buchanan et al.'s [29] algorithm that ensures the resulting vertex cut will be an a,b-separator.

Algorithm 6.3: a,b-Separator Algorithm (adapted from [29])

Data: A graph $G = (V, E)$, two vertices $a, b \in V$, and an a,b-separator $C \subset V$

Result: An inclusion-minimal a,b-separator $C' \subset C$

$C' = \{v \in C : \exists w \notin C, \{v, w\} \in E\}$;

$\mathcal{S} = \{S : S \text{ is a connected component of } G[V \backslash C']\}$;

for $v \in C'$ **do**

  if *v is not adjacent to $S_a, S_b \in \mathcal{S}$ with $a \in S_a$ and $b \in S_b$* **then**

    $C' = C' \backslash v$;

    Let $\mathcal{S}^v = \{S \in \mathcal{S}$: v is adjacent to $S\}$;

    Let $S_{new} = \bigcup_{S \in \mathcal{S}^v} S \cup v$;

    $\mathcal{S} = S_{new} \cup (\mathcal{S} \backslash \mathcal{S}^v)$;

  end

end

6.5 Computational Results for Connected Zero-Forcing

This section presents computational results from finding minimum connected zero-forcing sets using the methods previously mentioned.

6.5.1 Implementation Details

As with the basic zero-forcing problem, computational results were obtained on a Dell Precision T1650 workstation with a 3.3 GHz Intel Core i3-2120 CPU, 3.7 GB of RAM, and Red Hat Enterprise Linux version 6.6. The code was written in C++ and compiled with g++ version 4.8. Integer programs were solved using Gurobi version 5.5.0 set to use a single thread.

Model 6.8 was solved exactly like Model 6.2 with the addition of the MTZ constraints. Model 6.8 was solved in Gurobi with a POLLING callback to terminate the method after 2 hours. A maximal set of disjoint forts was added to the formulation before solving. This maximal set is found by iteratively finding minimum size forts (using Model 6.3) that are disjoint from each other until no more such forts can be found. Other fort constraints were added to the model using a MIPSOL callback to add violated forts. Gurobi calls this callback whenever it finds a new integral incumbent solution. The callback generates minimum size violated forts by using Gurobi to solve Model 6.3. If a violated fort is found, the MIPSOL callback adds that fort to the formulation as a lazy constraint. If a violated fort is not found, then Gurobi terminates with an optimal solution.

Model 6.9 was solved similarly to Model 6.8. A POLLING callback was used to terminate the method after 2 hours. Both fort constraints and a,b-separation inequalities are added to the model using a MIPSOL callback. This callback first generates a minimum size violated fort by using Gurobi to solve Model 6.3. If a

violated fort is found, the MIPSOL callback adds that fort to the formulation as a lazy constraint. If a violated fort is not found, then the callback checks whether the current solution is connected. This connectivity check is done using a breadth-first search to find a component of the graph induced by the current solution. If the solution is not connected, then Algorithm 6.3 is run on the separator given by all vertices that are not in the current solution. The a,b-separation inequality corresponding to the minimal separator given by Algorithm 6.3 is then added to Model 6.9 as a lazy constraint. If a violated fort is not found and the solution is connected, then Gurobi terminates with an optimal solution.

To enable lazy constraints, the "PreCrush" and "LazyConstraints" parameters were both set to 1. All the other parameters were left to their defaults in Gurobi. The branching strategy was also left to the Gurobi default. The times reported in this section are the time taken by Gurobi to optimize the relevant model. The time necessary for data input and setting up the Gurobi model is not reported.

6.5.2 Computational Tests

I again test the different algorithms on three classes of random graphs: cubic, connected Watts-Strogatz [133] graphs with parameters 5 and 0.3, and connected Watts-Strogatz graphs with parameters 10 and 0.3. The parameters for the Watts-Strogatz graphs refer to the number of neighbors initially given to each vertex and to the probability that an edge is rewired, respectively. I used my own C++ implementation to generate random cubic graphs, and I used the connected Watts-Strogatz graph generator from the NetworkX version 1.8.1 package in Python 2.7.6 to generate the Watts-Strogatz graphs. For each class of graph, I generated 5 random instances with 10, 20, ..., 100 vertices (I started at 20 vertices for

Watts-Strogatz graphs with parameters 10 and 0.3) and tested each algorithm until all 5 graphs of a certain size could not be solved by the algorithm within 2 hours.

Table 6.18 gives the number of instances of each type that were solved by each method. These results show that the Fort Cover IP with MTZ constraints (Model 6.8) performs better than the combinatorial brute force and branch and bound techinques as well as the Fort Cover IP with A,B separator constraints. This result is somewhat surprising since the A,B separator constraints have outperformed the MTZ constraints on other problems such as connected dominating set [29]. The results also show that the Fort Cover IP with MTZ constraints performs better for connected zero-forcing than Wavefront did for basic zero-forcing. Thus, the addition of the connectivity constraint makes the zero-forcing problem easier to solve.

The helpfulness of MTZ constraints can be clearly seen in the number of forts required to solve the Fort Cover IP with MTZ constraints vs. the plain Fort Cover IP. Table 6.16 compares the average number of forts required for the connected problem vs. the basic problem. The results show that the addition of MTZ constraints drastically decreases the number of forts that must be generated. The percentage of time used by the algorithm to generate forts is also decreased in the connected problem. Table 6.17 gives the average percentage of time spent by each method to generate forts.

Tables 6.20, 6.21, and 6.22 give the average running times on each graph size for each method on the cubic, Watts-Strogatz(5, 0.3), and Watts-Strogatz(10, 0,3) graphs, respectively. These results show that the branch and bound method performs best on small graphs, but as the size of the graphs increase or more connected subgraphs are contained in the graph, the Fort Cover IP with MTZ constraints starts to perform better than the branch and bound method. The Fort

Table 6.16 : Comparison of the number of forts required in connected vs. unconnected forcing.

	Cubic		WS (5,0.3)		WS (10,0.3)	
Graph Size	MTZ	FC	MTZ	FC	MTZ	FC
10	4.4	17.6	20.6	22.8	N/A	N/A
20	11.4	67.2	36.4	102.8	887.2	1133.8
30	9.4	153.2	79.8	245.2	6754.75	14024.8
40	8.4	1834.2	168.8	2334.2	T	T
50	14.6	4569	388.6	15801	T	T
60	45.6	27082	697.5	1489	T	T
70	65	T	554	T	T	T
80	178	T	T	T	T	T
90	156.8	T	T	T	T	

Note: The MTZ columns give the average number of forts required by the Fort Cover IP with MTZ constraints. The FC columns give the average number of forts required by the Fort Cover IP without the MTZ constraints. T indicates that the method did not solve within 2 hours for all of the instances of the specified size. The averages reported here are only over the instances that were solved by the methods within the 2 hour time limit.

Table 6.17 : Average percentage of time spent generating forts for connected zero-forcing.

	Cubic		WS (5,0.3)		WS (10,0.3)	
$\|V\|$	MTZ	A,B	MTZ	A,B	MTZ	A,B
10	21.9	29.6	43.6	57.6	N/A	N/A
20	32.3	47.7	34.7	62.7	33.0	75.3
30	6.7	24.8	36.7	63.5	6.9	24.8
40	0.8	7.9	12.8	27.9		
50	1.4	7.3	7.5	16.9		
60	1.9	4.1	3.1*	13.5*		
70	1.3	9.4	2.2*	14.3*		
80	0.3	7.8*				
90	0.6	7.1*				

Note: Asterisks indicate that not all instances of the specified size were solved within 2 hours. In these cases, the reported percentage is the percentage of the 2 hours that was used to generate forts.

Cover IP with MTZ constraints is able to solve the largest instances. The complete results for cubic, Watts-Strogatz (5, 0.3), and Watts-Strogatz (10, 0.3) graphs are given in Tables 6.23, 6.24, and 6.25, respectively.

Table 6.18 : Number of instances solved by each method.

Graph Type	Total	BF	B&B	IP with MTZ	IP with A,B
Cubic	50	15	40	**46**	36
WS (5, 0.3)	50	15	25	**30**	28
WS (10, 0.3)	45	5	5	9	**10**

Note: Columns BF and B&B are the brute force and branch and bound methods, respectively. Bold text indicates the method with the best performance.

Table 6.19 : Size of graphs where methods start to fail.

Graph Type	BF	B&B	IP with MTZ	IP with A,B
Cubic	40	70	**100**	80
WS (5, 0.3)	40	40	**60**	60
WS (10, 0.3)	30	30	30	**40**

Note: Columns BF and B&B are the brute force and branch and bound methods, respectively. Bold text indicates the method with the best performance.

Table 6.20 : Average running times for connected zero-forcing algorithms on random cubic graphs.

Graph Size	Avg. ZFS size	Brute Force	Branch & Bound	IP with MTZ	IP with A,B
10	3.8	0.007	**0.001**	0.02	0.01
20	5.4	0.50	**0.0128**	0.12	0.09
30	7.8	1334.21	**0.28**	0.76	0.49
40	9.8	T	537.0	4.49	**3.28**
50	10.4	T	21.97	**19.95**	79.79
60	12	T	550.0	**56.60**	1141.84
70	13.4	T	287.33*	**261.54**	3541.51
80	15.6	T	2857.07*	**1869.82**	1685.30*
90	15.2	T	3192.58*	**2952.18**	T
100	16	T	T	**2566.91***	T

Note: Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported results is the average time for the instances that were successfully solved. Bold text indicates the method with the best performance.

Table 6.21 : Average running times for connected zero-forcing algorithms on random Watts-Strogatz graphs with parameters (5, 0.3).

Graph Size	Avg. ZFS size	Brute Force	Branch & Bound	IP with MTZ	IP with A,B
10	4.4	0.011	**0.003**	0.07	0.05
20	6.2	1.72	**0.21**	0.42	0.24
30	7	90.62	**1.34**	2.73	2.15
40	9.4	T	25.86*	25.47	**20.48**
50	10.8	T	1102.20*	**236.47**	267.90
60	11.75	T	2223.9*	**2199.54***	1876.90*
70	13	T	T	**7180.12***	T

Note: Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported results is the average time for the instances that were successfully solved. Bold text indicates the method with the best performance (measured first by number of instances of the relevant size solved and then by average time).

Table 6.22 : Average running times for connected zero-forcing algorithms on random Watts-Strogatz graphs with parameters (10, 0.3).

Graph Size	Avg. ZFS size	Brute Force	Branch & Bound	IP with MTZ	IP with A,B
20	12	112.63	16.49	26.21	**9.65**
30	15.4	T	T	3768.18*	**850.94**
40		T	T	T	T

Note: Asterisks indicate that not all instances of the specified size were solved. In these cases, the reported results is the average time for the instances that were successfully solved. Bold text indicates the method with the best performance.

Table 6.23 : Running times for connected zero-forcing algorithms on cubic graphs.

		BF	B & B	IP with MTZ			IP with A,B				
Graphs	Z(G)	time	time	time	forts	fort time	time	forts	fort time	AB cuts	AB time
cubic10_1	4	0.009	**0.001**	0.013	2	0.0003	0.011	2	0.003	5	0.001
cubic10_2	4	0.012	**0.001**	0.017	2	0.001	0.009	2	0.002	1	0.000
cubic10_3	4	0.007	**0.001**	0.027	6	0.013	0.010	5	0.003	0	0.000
cubic10_4	4	0.007	**0.001**	0.016	3	0.001	0.007	3	0.002	3	0.000
cubic10_5	3	0.001	**0.001**	0.015	9	0.007	0.010	8	0.004	0	0.000
cubic20_1	5	0.33	**0.007**	0.111	11	0.052	0.077	10	0.050	0	0.000
cubic20_2	6	0.96	**0.021**	0.093	6	0.004	0.063	15	0.006	9	0.002
cubic20_3	6	0.87	**0.019**	0.152	18	0.048	0.12	26	0.059	2	0.001
cubic20_4	5	0.16	**0.009**	0.131	11	0.041	0.078	12	0.046	0	0.000
cubic20_5	5	0.19	**0.008**	0.099	11	0.047	0.088	10	0.049	1	0.000
cubic30_1	10	5301.21	**0.912**	1.70	5	0.002	1.00	31	0.040	18	0.006
cubic30_2	6	13.91	**0.023**	0.25	6	0.016	0.16	17	0.080	0	0.001
cubic30_3	7	41.17	**0.067**	0.71	24	0.139	1.03	27	0.13	0	0.001
cubic30_4	9	1170.48	**0.326**	0.85	4	0.002	0.16	16	0.041	19	0.006
cubic30_5	7	144.3	**0.084**	0.31	8	0.022	0.11	13	0.035	8	0.003
cubic40_1	11	T	10.15	8.98	8	0.013	**2.51**	43	0.18	49	0.022
cubic40_2	9	T	2511.15	1.52	4	0.012	**0.80**	30	0.16	19	0.011
cubic40_3	9	T	160.36	**2.06**	11	0.053	3.42	31	0.19	6	0.004
cubic40_4	10	T	**1.89**	4.64	10	0.024	6.83	42	0.14	23	0.011
cubic40_5	10	T	**1.43**	5.26	9	0.010	2.86	26	0.13	9	0.005
cubic50_1	10	T	63.75	**7.92**	10	0.060	19.05	195	2.53	64	0.039
cubic50_2	11	T	**11.71**	37.28	14	0.116	38.55	82	0.996	27	0.018
cubic50_3	9	T	**1.84**	4.98	13	0.254	24.76	123	2.98	17	0.012
cubic50_4	12	T	**27.06**	36.67	22	0.091	293.73	351	2.86	76	0.047
cubic50_5	10	T	**5.49**	12.88	14	0.091	22.86	117	1.74	13	0.009
cubic60_1	11		**14.7**	48.84	72	1.75	475.44	700	30.89	4	0.005
cubic60_2	15		2659.64	**96.29**	17	0.11	4033.24	1213	21.03	121	0.111
cubic60_3	11		**11.48**	39.88	58	1.10	731.66	1155	47.51	8	0.008
cubic60_4	11		**15.2**	36.17	51	0.75	207.21	233	6.19	6	0.006
cubic60_5	12		**48.99**	61.81	30	0.45	261.63	413	10.00	20	0.018
cubic70_1	13		**127.73**	200.52	43	1.04	7019.35	14773	1278.55	50	0.057
cubic70_2	14		493.68	**278.79**	55	1.31	5360.54	7240	447.52	19	0.021
cubic70_3	12		**73.97**	207.96	154	10.84	2291.46	2036	158.86	17	0.020
cubic70_4	14		453.94	**300.41**	39	0.90	1880.15	3006	149.05	119	0.125
cubic70_5	14		T	**320.02**	34	0.61	1156.06	1687	67.19	130	0.130
cubic80_1	15		1199.35	**1001.07**	94	3.77	T	5279	395.75	63	0.085
cubic80_2	16		T	**1463.63**	84	2.18	T	4938	208.18	324	0.394
cubic80_3	17		T	**5163.61**	608	34.61	T	10923	890.99	68	0.098
cubic80_4	16		4514.79	**1188.15**	49	1.39	T	12015	742.84	353	0.453
cubic80_5	14		T	**532.63**	55	1.74	1685.30	1246	133.00	66	0.084
cubic90_1	14		**830.4**	1550.05	142	9.03	T	2406	535.63	3	0.009
cubic90_2	14		3066.68	**900.06**	158	11.2	T	1456	227.56	15	0.027
cubic90_3	15		**2479.32**	4095.76	227	16.55	T	1763	442.76	28	0.047
cubic90_4	17		T	**5357.49**	100	3.97	T	5227	786.14	16	0.029
cubic90_5	16		6393.93	**2857.52**	157	17.69	T	2868	574.38	73	0.116
cubic100_1			T	T	141	5.69					
cubic100_2			T	T	267	21.36					
cubic100_3			T	T	154	10.79					
cubic100_4	16		T	**2566.91**	100	16.05					
cubic100_5			T	T	653	92.4					

Note: All times are in seconds. The time columns give the total time required. The fort time and AB time columns give the time used to generate forts and a,b-separation constraints, respectively. The forts and AB cuts columns give the number of forts or a,b-constraints generated, respectively. T indicates the algorithm did not find a solution within 2 hours. Bold text indicates the method with the best performance.

Table 6.24 : Running times for connected zero-forcing algorithms on random Watts-Strogatz graphs with parameters (5, 0.3).

		BF	B & B	IP with MTZ			IP with A,B				
Graphs	Z(G)	time	time	time	forts	fort time	time	forts	fort time	AB cuts	AB time
WS10_5_0.3_1	4	0.011	**0.002**	0.064	15	0.034	0.038	14	0.025	0	0.000
WS10_5_0.3_2	4	0.007	**0.002**	0.099	14	0.017	0.053	14	0.020	0	0.000
WS10_5_0.3_3	4	0.008	**0.002**	0.058	13	0.023	0.049	14	0.023	0	0.000
WS10_5_0.3_4	5	0.017	**0.005**	0.067	29	0.034	0.061	30	0.040	0	0.000
WS10_5_0.3_5	5	0.011	**0.005**	0.080	32	0.046	0.068	30	0.049	0	0.000
WS20_5_0.3_1	6	2.05	**0.040**	0.391	44	0.156	0.25	46	0.17	1	0.000
WS20_5_0.3_2	6	0.89	0.778	0.335	22	0.094	**0.17**	27	0.09	7	0.002
WS20_5_0.3_3	6	0.85	**0.043**	0.354	38	0.181	0.23	41	0.17	0	0.000
WS20_5_0.3_4	7	3.05	**0.087**	0.639	45	0.091	0.33	75	0.16	0	0.000
WS20_5_0.3_5	6	1.74	**0.094**	0.388	33	0.155	0.20	29	0.14	0	0.000
WS30_5_0.3_1	6	29.99	**0.198**	1.58	55	0.945	1.25	61	1.05	0	0.001
WS30_5_0.3_2	8	182.18	**2.05**	5.47	160	1.553	5.29	210	2.04	0	0.001
WS30_5_0.3_3	7	50.97	**0.283**	2.63	52	0.602	1.37	74	0.92	0	0.000
WS30_5_0.3_4	7	136.59	3.81	3.76	66	0.693	**1.61**	76	0.92	0	0.001
WS30_5_0.3_5	7	53.35	**0.346**	1.78	66	0.728	1.22	71	0.86	6	0.002
WS40_5_0.3_1	9	T	29.21	16.39	124	2.51	**14.32**	233	4.75	1	0.002
WS40_5_0.3_2	9	T	**14.00**	28.30	221	4.59	28.01	318	6.91	1	0.002
WS40_5_0.3_3	10	T	T	23.26	101	1.33	**13.39**	190	2.92	7	0.005
WS40_5_0.3_4	10	T	43.0	33.97	202	2.89	**26.64**	470	5.90	1	0.002
WS40_5_0.3_5	9	T	**17.24**	25.43	196	4.60	20.06	273	7.60	0	0.001
WS50_5_0.3_1	11		3586.52	**140.23**	326	12.99	288.44	966	27.24	14	0.011
WS50_5_0.3_2	11		196.02	**168.79**	183	4.97	268.99	1170	34.78	24	0.016
WS50_5_0.3_3	10		T	**78.06**	218	9.67	132.52	916	30.83	12	0.009
WS50_5_0.3_4	11		324.67	**252.13**	521	18.66	259.45	1336	50.15	1	0.002
WS50_5_0.3_5	11		**301.57**	543.16	695	30.49	390.11	1869	75.33	6	0.005
WS60_5_0.3_1	12		3684.89	3058.47	832	61.92	**3017.09**	5813	394.68	37	0.035
WS60_5_0.3_2	13		T	**4724.15**	1392	94.59	T	12962	698.46	8	0.009
WS60_5_0.3_3	11		T	103.84	77	4.53	**77.27**	227	9.0	40	0.035
WS60_5_0.3_4			T	T	2546	215.84	T	13514	1035.18	5	0.007
WS60_5_0.3_5	11		762.91	911.69	489	35.59	**659.43**	1580	124.49	4	0.007
WS70_5_0.3_1			T	T	1718	179.09	T	12723	1453.47	8	0.012
WS70_5_0.3_2	13		T	**7180.12**	554	76.96	T	4440	645.31	22	0.025
WS70_5_0.3_3			T	T	1147	149.10	T	9974	1205.51	7	0.010
WS70_5_0.3_4			T	T	1454	139.07	T	6601	768.98	16	0.019
WS70_5_0.3_5			T	T	2321	247.83	T	9490	1071.59	25	0.029
WS80_5_0.3_1			T	T	1466	341.49					
WS80_5_0.3_2			T	T							
WS80_5_0.3_3			T	T							
WS80_5_0.3_4			T	T							
WS80_5_0.3_5			T	T							
WS90_5_0.3_1			T								
WS90_5_0.3_2			T								
WS90_5_0.3_3			T								
WS90_5_0.3_4			T								
WS90_5_0.3_5			T								

Note: All times are in seconds. The time columns give the total time required. The fort time and AB time columns give the time used to generate forts and a,b-separation constraints, respectively. The forts and AB cuts columns give the number of forts or a,b-constraints generated, respectively. T indicates the algorithm did not find a solution within 2 hours. Bold text indicates the method with the best performance.

Table 6.25 : Running times for connected zero-forcing algorithms on random Watts-Strogatz graphs with parameters (10, 0.3).

		BF	B & B	IP with MTZ			IP with A,B				
Graphs	Z(G)	time	time	time	forts	fort time	time	forts	fort time	AB cuts	AB time
WS20_10_0.3_1	12	114.64	16.67	24.35	854	9.19	**9.76**	770	7.47	0	0.001
WS20_10_0.3_2	12	111.95	16.4	29.16	938	9.25	**10.30**	864	7.76	0	0.001
WS20_10_0.3_3	12	112.21	16.54	25.49	906	8.70	**9.34**	824	7.08	0	0.001
WS20_10_0.3_4	12	112.03	16.48	27.54	869	7.64	**9.40**	814	6.94	0	0.001
WS20_10_0.3_5	12	112.33	16.37	24.54	869	8.30	**9.47**	811	7.10	0	0.001
WS30_10_0.3_1	15	T	T	2472.75	4773	136.23	**662.27**	4938	132.91	0	0.002
WS30_10_0.3_2	16	T	T	6017.80	9828	244.39	**1118.27**	9048	195.06	0	0.001
WS30_10_0.3_3	14	T	T	786.95	3332	132.75	**190.57**	2708	98.74	0	0.002
WS30_10_0.3_4	16	T	T	5795.21	9086	236.75	**1144.46**	8384	190.98	0	0.002
WS30_10_0.3_5	16	T	T	T	9922	281.13	**1139.14**	8595	204.86	0	0.001
WS40_10_0.3_1				T	13169	826.00	T	20305	1196.88	0	0.002
WS40_10_0.3_2				T	8472	672.62	T	12833	792.71	0	0.001
WS40_10_0.3_3				T	9721	803.86	T	12988	1000.30	0	0.002
WS40_10_0.3_4				T	10461	801.25	T	15714	1093.75	0	0.003
WS40_10_0.3_5				T	9308	726.67	T	14422	1000.56	0	0.003

Note: All times are in seconds. The time columns give the total time required. The fort time and AB time columns give the time used to generate forts and a,b-separation constraints, respectively. The forts and AB cuts columns give the number of forts or a,b-constraints generated, respectively. T indicates the algorithm did not find a solution within 2 hours. Bold text indicates the method with the best performance.

6.6 Conclusions

This chapter has introduced new methods for computing minimum zero-forcing sets of graphs. The computational results show that integer programming models based on a fort covering perspective perform much better than a model based on the standard infection perspective. In some cases, the fort covering integer program can be improved by only adding forts that are facet-inducing; however, this improvement is not significant enough to increase the size of problems that can be solved by the integer program. While the fort covering perspective is an improvement over the infection perspective, the C++ implementation of the Wavefront algorithm developed for this chapter can solve larger problems than the integer programs and also solves the problems in less time. Thus, the Wavefront

algorithm is the best algorithm for the basic zero-forcing problem. However, some graphs, such as stars, are difficult for the Wavefront algorithm, but easy to solve with the Fort Cover integer program.

While the Wavefront algorithm is best for the basic zero-forcing problem, the integer programming methods allow the addition of different types of constraints, such as connectivity. Although the combinatorial branch and bound method presented in this chapter performs well for small problems, it does not scale well to larger problems. It relies on generating the connected induced subgraphs that are no larger than the zero forcing number. As graphs get larger and have higher degrees, they will have more such subgraphs, and the branch and bound method will have to generate more subgraphs. For the connected zero-forcing problem, the fort covering based integer program with either MTZ constraints or a,b-separation constraints is able to solve larger instances than combinatorial methods. On graphs with at least 40 vertices, these integer programs are also faster on average than the branch and bound method.

Between the two integer programs, the MTZ constraints are better for the cubic graphs, which are relatively sparse. For the Watts-Strogatz graphs with parameters 5 and 0.3, the methods showed similar performance. The a,b-separation constraints are better for the Watts-Strogatz graphs with parameters 10 and 0.3. As the average degree of the vertices increases, the likelihood that a chosen subset of vertices will induce a connected graph increases. Therefore, for the Watts-Strogatz graphs with high average degree (paramers 10 and 0.3), the a,b-separation inequalities were usually not necessary. In such cases, the a,b-separation inequalities are not added to the model, and it is solved as a basic zero-forcing problem. In general for the connected problem, the fort covering integer program with MTZ

constraints is the best algorithm.

Both in the connected and basic versions, the zero-forcing problem is difficult at least in part because of the symmetry of solutions. This symmetry does not arise from single vertices being indistinguishable, but rather from sets of vertices being indistinguishable. For each solution, an equivalent solution can be obtained by simply choosing the end vertices of each forcing chain [17]. This symmetry is harder to detect than simple isomporphisms in the graph. However, any method for dealing with the symmetry of zero-forcing problems has the potential to drastically improve the performance of the integer programs presented in this chapter. Therefore, future work into this problem should focus on breaking this symmetry.

Chapter 7

Conclusions

This thesis has presented improved methods for solving two graph location problems, the p-Median problem and the zero-forcing problem. I also presented methods for solving the connected variant of the zero-forcing problem. This thesis also introduced new bounds on the size of minimum zero-forcing sets and on the zero-forcing iteration index.

For the p-median problem, this thesis gave a new algorithm based on branch decompositions of linear programming or heuristic support graphs. The BDPM algorithm finds the best solution whose edges are a subset of the edges of the input support graph, and it is a type of exact heuristic concentration. The algorithm run on a linear programming support graph, BDPM-LP, proved to be an effective technique to find a high quality integral solution when a branch decomposition of the linear programming support graph could be found with a width no more than 7. It is more accurate than the Imp-GA algorithm as long as the number of medians is not very small. It is also more accurate than the HHP algorithm when the linear program does not have to be altered because the width of its branch decomposition is too high.

The version of BDPM that is run on a pool of heuristic solutions, BDPM-H, was able to produce improved solutions from the heuristic pool using GRASP as the heuristic. Larger p values generally lead to more significant improvements. BDPM-GRASP outperforms Imp-GA when the branch decompositions have low

width (no more than 7 in our experiments). It is less accurate on average than HHP. However, the difference is not large, and, in some cases, BDPM-GRASP was more accurate. BDPM-GRASP is also competitive in running time with HHP when branchwidths are no more than 5.

The performance of BDPM-H is dependent on the heuristic, or heuristics, chosen to form the support graph. An interesting future research direction would be to investigate whether certain classes of heuristics allow for more improvement or better performance than the GRASP heuristic. Another interesting direction, since HHP already uses GRASP as a base step, would be to combine BDPM-GRASP as a subroutine in HHP.

In general, HHP is still a better general algorithm for the p-median problem than either BDPM-GRASP or BDPM-LP. However, when the linear program is easy to solve and the support graph has a low width decomposition, the BDPM-LP algorithm is better than HHP. Since, a heuristic method for finding branch decompositions is relatively fast compared to either BDPM-LP or HHP, very little extra compuational time is needed to check whether a low width decomposition is available. Thus, a potential way to use BDPM is to run the heuristics or the linear program to get a support graph. Then, check whether a low width decomposition is found by a heuristic. If a low width decomposition is found, then use BDPM; otherwise, use a heuristic such as HHP.

For the zero-forcing problem, this thesis presented both theoretical and computational results. For theoretical results, I showed that branchwidth is a tight lower bound on the zero-forcing number. I also introduced the concept of zero-forcing forts and showed that every zero-forcing set must contain a vertex in each zero-forcing fort. The fort perspective on zero-forcing is useful both for

theoretical and computational results. For example, it was used in this thesis to prove the correctness of the Wavefront algorithm, and it was also used to model the zero-forcing problem as an integer program.

In addition to the fort theory, I showed that the zero-forcing iteration index of a cubic graph is at most $\frac{3|V|}{4}$. This bound is the first non-trivial bound on the zero-forcing iteration index. Although the $\frac{3|V|}{4}$ bound is tight in the sense that a minimum zero-forcing set can be found for certain graphs that takes $\frac{3|V|}{4}$ iterations to force the graph, it is my conjecture that such graphs have a minimum forcing set that will force the graph in fewer iterations. Thus, a future direction of research is to improve on the $\frac{3|V|}{4}$ bound.

This thesis also introduced new ways to compute minimum zero-forcing sets and minimum connected zero-forcing sets of a graph. I gave an integer programming formulation that was based on the infection or color-change rule definition of the zero-forcing process; however, this modeling perspective requires big-M constraints to model the infection process. These constraints lead to poor performance for the integer program. I also used the theory of zero-forcing forts to develop another integer programming model based on covering all forts in the graph. Although this method requires separation of violated fort constraints, it still performs better than the infection perspective. Both of these integer programming models do not perform better than the existing Wavefront algorithm [35]; however, the Wavefront algorithm is limited to the basic zero-forcing problem. If additional constraints are added, such as requiring connectivity of the zero-forcing set, then the Wavefront algorithm is no longer applicable. However, the integer programming approaches can easily be adapted to the connected problem.

This thesis introduced two methods for computing minimum connected

zero-forcing sets of a graph. The first method is a combinatorial branch and bound style method based on generating all connected subsets of vertices, but pruning branches of the search tree that lead to connected subsets larger than the current best connected zero-forcing set found up to that point. This method works well when there is a connected forcing set of small size and many branches of the search tree are pruned quickly. However, it does not work well when the graph has a relatively large number of connected subgraphs that are smaller than the zero-forcing set because the branch and bound method must generate all of these connected subgraphs.

The second method is an integer programming method that simply adds connectivity constraints, such as Miller-Tucker-Zemlin (MTZ) constraints [98] or a,b-separation constraints (see for example [29], [72], and [38]), to the fort covering model of the basic problem. This method requires no special properties of the graph under consideration and, on cubic graphs, gives performance better than that of Wavefront on the basic problem. It also beats the branch and bound method in the size of graphs for which connected zero-forcing sets can be computed and is faster than the branch and bound method for graphs with at least 40 vertices. Of the two methods for enforcing connectivity, the MTZ constraints seem to perform better for sparser graphs and a,b-separation performs better on denser graphs.

For difficult instances of both the basic and the connected zero-forcing problem, the integer program has difficulty proving optimality, and the lower bound eventually increases very slowly. Thus, a good direction for future research is to investigate more facets of the zero-forcing polytope or determine methods for findings forts that will contribute significantly to the lower bound. Additionally, in my computiational experiments, I used integer programs to separate violated fort

constraints. A faster combinatorial method for separating these constraints would be interesting.

The most difficult aspect of computing zero-forcing sets seems to be dealing with the symmetry inherent in the basic problem. Given any zero-forcing set, an equivalent zero-forcing set can be found by simply reversing the direction that the infection travels through the graph [17]. This fact leads to multiple equivalent solutions and unnecessary repetition of work by the solver. Unfortunately, this symmetry is not immediately apparent as isomorphisms in the graph. So, it is not immediately apparent how the methods built for dealing with isomorphic variables in an integer program (see for example, [96]) can be applied to deal with the symmetry of the zero-forcing problem. Therefore, finding a way to break the symmetry of the problem, either through constraints in the integer program model or through branching rules, would be a valuable direction of future research.

Bibliography

[1] A. Aazami, *Hardness results and approximation algorithms for some problems on graphs*, PhD thesis, University of Waterloo, 2008.

[2] ——, *Domination in graphs with bounded propagation: algorithms, formulations and hardness results*, Journal of Combinatorial Optimization, 19 (2010), pp. 429–456.

[3] AIM Minimum Rank Special Graphs Work Group, *Zero forcing sets and the minimum rank of graphs*, Linear Algebra and its Applications, 428 (2008), pp. 1628 – 1648.

[4] E. Almodovar, L. DeLoss, L. Hogben, K. Hogenson, K. Murphy, T. Peters, and C. A. Ramírez, *Minimum rank, maximum nullity and zero forcing number for selected graph families*, Involve, 3 (2010), pp. 371–392.

[5] O. Alp, E. Erkut, and Z. Drezner, *An efficient genetic algorithm for the p-median problem*, Annals of Operations Research, 122 (2003), pp. 21–42.

[6] D. Amos, Y. Caro, R. Davila, and R. Pepper, *Upper bounds on the k-forcing number of a graph*, Discrete Applied Mathematics, 181 (2015), pp. 1 – 10.

[7] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*, Princeton University

Press, Princeton, 2006.

[8] J. E. C. Arroyo, M. dos Santos Soares, and P. M. dos Santos, *A grasp heuristic with path-relinking for a bi-objective p-median problem*, in 2010 10th International Conference on Hybrid Intelligent Systems, Aug 2010, pp. 97–102.

[9] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit, *Local search heuristics for k-median and facility location problems*, SIAM Journal on Computing, 33 (2004), pp. 544–562.

[10] P. Avella, M. Boccia, S. Salerno, and I. Vasilyev, *An aggregation heuristic for large scale p-median problem*, Computers & Operations Research, 39 (2012), pp. 1625 – 1632.

[11] P. Avella, M. Boccia, and I. Vasilyev, *Computational experience with general cutting planes for the set covering problem*, Operations Research Letters, 37 (2009), pp. 16 – 20.

[12] P. Avella and A. Sassano, *On the p-median polytope*, Mathematical Programming, 89 (2001), pp. 395–411.

[13] P. Avella, A. Sassano, and I. Vasil'ev, *Computational study of large-scale* p*-median problems*, Mathematical Programming, 109 (2007), pp. 89–114.

[14] D. Avis and K. Fukuda, *Reverse search for enumeration*, Discrete Applied Mathematics, 65 (1996), pp. 21 – 46.

[15] K. Baker, *A heuristic approach to locating a fixed number of facilities*, Logistics and Transportation Review, 10 (1974), pp. 195–205.

[16] E. Balas and S. M. Ng, *On the set covering polytope: I. all the facets with coefficients in {0, 1, 2}*, Mathematical Programming, 43 (1989), pp. 57–69.

[17] F. Barioli, W. Barrett, S. M. Fallat, H. T. Hall, L. Hogben, B. Shader, P. van den Driessche, and H. van der Holst, *Zero forcing parameters and minimum rank problems*, Linear Algebra and its Applications, 433 (2010), pp. 401 – 411.

[18] ——, *Parameters related to tree-width, zero forcing, and maximum nullity of a graph*, Journal of Graph Theory, 72 (2013), pp. 146–177.

[19] W. Barret, S. Butler, M. Catral, S. M. Fallat, and H. T. Hall, *The maximum nullity of a complete subdivision graph is equal to its zero forcing number*, Electronic Journal of Linear Algebra, 27 (2014).

[20] J. E. Beasley, *A note on solving large p-median problems*, European Journal of Operational Research, 21 (1985), pp. 270–273.

[21] ——, *Lagrangean heuristics for location problems*, European Journal of Operational Research, 65 (1993), pp. 383 – 399.

[22] A. Berliner, C. Brown, J. Carlson, N. Cox, L. Hogben, J. Hu, K. Jacobs, K. Manternach, T. Peters, N. Warnberg, and M. Young, *Path cover number, maximum nullity, and zero forcing number of oriented graphs and other simple digraphs*, Involve, 8 (2015), pp. 147–167.

[23] B. Bozkaya, J. Zhang, and E. Erkut, *An efficient genetic algorithm for the p-median problem*, in Facility location: Applications and theory, Z. Drezner and H. W. Hamacher, eds., Springer, Berlin, 2002, ch. 6, pp. 179–205.

[24] O. Briant and D. Naddef, *The optimal diversity management problem*, Operations Research, 52 (2004), pp. 515–526.

[25] B. Brimkov, *Complexity and Computation of Connected Zero Forcing*, ArXiv e-prints, (2016).

[26] B. Brimkov and R. Davila, *Characterizations of the Connected Forcing Number of a Graph*, ArXiv e-prints, (2016).

[27] B. Brimkov, C. C. Fast, and I. V. Hicks, *Computational Approaches for Zero Forcing and Related Problems*, Working Paper, (2017).

[28] ——, *Graphs with Extremal Connected Forcing Numbers*, ArXiv e-prints, (2017).

[29] A. Buchanan, J. S. Sung, S. Butenko, and E. L. Pasiliao, *An integer programming approach for fault-tolerant connected dominating sets*, INFORMS Journal on Computing, 27 (2015), pp. 178–188.

[30] D. Burgarth, D. D'Alessandro, L. Hogben, S. Severini, and M. Young, *Zero forcing, linear and quantum controllability for systems evolving on networks*, IEEE Transactions on Automatic Control, 58 (2013), pp. 2349–2354.

[31] D. Burgarth and V. Giovannetti, *Full control by locally induced relaxation*, Physical Review Letters, 99 (2007), p. 100501.

[32] D. Burgarth, V. Giovannetti, L. Hogben, S. Severini, and M. Young, *Logic circuits from zero forcing*, Natural Computing, 14 (2015), pp. 485–490.

[33] D. Burgarth and K. Maruyama, *Indirect hamiltonian identification through a small gateway*, New Journal of Physics, 11 (2009), p. 103019.

[34] D. Burgarth, K. Maruyama, and F. Nori, *Indirect quantum tomography of quadratic hamiltonians*, New Journal of Physics, 13 (2011), p. 013019.

[35] S. Butler, L. DeLoss, J. Grout, H. T. Hall, J. LaGrange, T. McKay, J. Smith, and G. Tims., *Minimum Rank Library (Sage programs for calculating bounds on the minimum rank of a graph, and for computing zero forcing parameters)*, 2014. `https://github.com/jasongrout/minimum_rank`.

[36] S. Butler and M. Young, *Throttling zero forcing propagation speed on graphs*, The Australasian Journal of Combinatorics, 57 (2013), pp. 65–71.

[37] M. E. Captivo, *Fast primal and dual heuristics for the p-median location problem*, European Journal of Operational Research, 52 (1991), pp. 65 – 74.

[38] R. Carvajal, M. Constantino, M. Goycoolea, J. P. Vielma, and A. Weintraub, *Imposing connectivity constraints in forest planning models*, Operations Research, 61 (2013), pp. 824–836.

[39] M. Catral, A. Cepek, L. Hogben, M. Huynh, K. Lazebnik, T. Peters, and M. Young, *Zero forcing number, maximum nullity, and path cover number of subdivided graphs*, Electronic Journal of Linear Algebra, 23 (2012).

[40] K. B. Chilakamarri, N. Dean, C. X. Kang, and E. Yi, *Iteration index of a zero forcing set in a graph*, Bulletin of the Institute of Combinatorial Mathematics and its Applications, 64 (2012), pp. 57–72.

[41] W. Cook and P. D. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing, (2003), pp. 233–248.

[42] G. Cornuejols, M. L. Fisher, and G. L. Nemhauser, *Exceptional paper—Location of bank accounts to optimize float: An analytic study of exact and approximate algorithms*, Management Science, 23 (1977), pp. 789–810.

[43] G. Cornuejols, G. L. Nemhauser, and L. A. Wolsey, *A canonical representation of simple plant location problems and its applications*, SIAM Journal on Algebraic Discrete Methods, 1 (1980), pp. 261–272.

[44] E. S. Correa, M. T. A. Steiner, A. A. Freitas, and C. Carnieri, *A genetic algorithm for solving a capacitated p-median problem*, Numerical Algorithms, 35 (2004), pp. 373–388.

[45] G. Dantzig, R. Fulkerson, and S. Johnson, *Solution of a large-scale traveling-salesman problem*, Operations Research, 2 (1954), pp. 393–410.

[46] R. Davila, *Bounding the forcing number of a graph*, Master's thesis, Rice University, 2015.

[47] R. Davila and M. Henning, *Total Forcing Sets in Trees*, ArXiv e-prints, (2017).

[48] R. Davila, M. Henning, C. Magnant, and R. Pepper, *Bounds on the connected forcing number of a graph*, ArXiv e-prints, (2016).

[49] R. Davila and M. A. Henning, *On the Total Forcing Number of a Graph*, ArXiv e-prints, (2017).

[50] R. Davila and F. Kenter, *Bounds for the Zero-Forcing Number of Graphs with Large Girth*, Theory and Applications of Graphs, 2 (2015), pp. 1–10.

[51] P. J. Densham and G. Rushton, *Strategies for solving large location-allocation problems by heuristic methods*, Environment and Planning A, 24 (1992), pp. 289–304.

[52] C. Dibble and P. J. Densham, *Generating interesting alternatives in gis and sdss using genetic algorithms*, in GIS/LIS Proceedings, vol. 2, American Society for Photogrammetry and Remote Sensing, November 1993, pp. 180–189.

[53] P. A. Dreyer Jr. and F. S. Roberts, *Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion*, Discrete Applied Mathematics, 157 (2009), pp. 1615 – 1627.

[54] Z. Drezner, *Dynamic facility location: The progressive p-median problem*, Location Science, 3 (1995), pp. 1 – 7.

[55] ——, *On the conditional p-median problem*, Computers & Operations Research, 22 (1995), pp. 525 – 530.

[56] O. du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen, *Stabilized column generation*, Discrete Mathematics, 194 (1999), pp. 229 – 237.

[57] C. J. Edholm, L. Hogben, M. Huynh, J. LaGrange, and D. D. Row, *Vertex and edge spread of zero forcing number, maximum nullity, and*

minimum rank of a graph, Linear Algebra and its Applications, 436 (2012), pp. 4352 – 4372. Special Issue on Matrices Described by Patterns.

[58] J. Ekstrand, C. Erickson, H. T. Hall, D. Hay, L. Hogben, R. Johnson, N. Kingsley, S. Osborne, T. Peters, J. Roat, A. Ross, D. D. Row, N. Warnberg, and M. Young, *Positive semidefinite zero forcing*, Linear Algebra and its Applications, 439 (2013), pp. 1862 – 1874.

[59] J. Ekstrand, C. Erickson, D. Hay, L. Hogben, and J. Roat, *Note on positive semidefinite maximum nullity and positive semidefinite zero forcing number of partial 2-trees*, Electronic Journal of Linear Algebra, 23 (2012).

[60] S. Elloumi, *A tighter formulation of the p-median problem*, Journal of Combinatorial Optimization, 19 (2010), pp. 69–83.

[61] D. Erlenkotter, *A dual-based procedure for uncapacitated facility location*, Operations Research, 26 (1978), pp. 992–1009.

[62] L. Eroh, C. X. Kang, and E. Yi, *Metric dimension and zero forcing number of two families of line graphs*, Mathematica Bohemica, 139 (2014), pp. 467–483.

[63] ——, *On zero forcing number of graphs and their complements*, Discrete Mathematics, Algorithms and Applications, 07 (2015), p. 1550002.

[64] ——, *A comparison between the metric dimension and zero forcing number of trees and unicyclic graphs*, Acta Mathematica Sinica, English Series, (2017), pp. 1–17.

[65] S. Fallat, K. Meagher, and B. Yang, *On the complexity of the positive semidefinite zero forcing number*, Linear Algebra and its Applications, 491 (2016), pp. 101 – 122. Proceedings of the 19th {ILAS} Conference, Seoul, South Korea 2014.

[66] N. Fan and J.-P. Watson, *Solving the connected dominating set problem and power dominating set problem by integer programming*, in Combinatorial Optimization and Applications: 6th International Conference, COCOA 2012, Banff, AB, Canada, August 5-9, 2012. Proceedings, G. Lin, ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 371–383.

[67] C. C. Fast and I. V. Hicks, *The effect of vertex degrees on the zero-forcing number and iteration index of a graph*, Submitted, (2016).

[68] ——, *A branch decomposition algorithm for the p-median problem*, To appear in INFORMS Journal on Computing, (2017).

[69] E. Feldman, F. A. Lehrer, and T. L. Ray, *Warehouse location under continuous economies of scale*, Management Science, 12 (1966), pp. 670–684.

[70] T. A. Feo and M. G. C. Resende, *A probabilistic heuristic for a computationally difficult set covering problem*, Operations Research Letters, 8 (1989), pp. 67 – 71.

[71] ——, *Greedy randomized adaptive search procedures*, Journal of Global Optimization, 6 (1995), pp. 109–133.

[72] M. Fischetti, M. Leitner, I. Ljubić, M. Luipersbeck, M. Monaci, M. Resch, D. Salvagnin, and M. Sinnl, *Thinning out steiner trees: a*

node-based model for uniform edge costs, Mathematical Programming Computation, (2016), pp. 1–27.

[73] M. Fischetti and A. Lodi, *Optimizing over the first chvátal closure*, Mathematical Programming, 110 (2007), pp. 3–20.

[74] G. Fung and O. L. Mangasarian, *Semi-supervised support vector machines for unlabeled data classification*, Optimization Methods and Software, 15 (2001), pp. 29–44.

[75] S. García, M. Labbé, and A. Marín, *Solving large p-median problems with a radius formulation*, INFORMS Journal on Computing, 23 (2011), pp. 546–556.

[76] R. S. Garfinkel, A. W. Neebe, and M. R. Rao, *An algorithm for the m-median plant location problem*, Transportation Science, 8 (1974), pp. 217–236.

[77] M. Gentner, L. D. Penso, D. Rautenbach, and U. S. Souza, *Extremal values and bounds for the zero forcing number*, Discrete Applied Mathematics, 214 (2016), pp. 196 – 200.

[78] S. L. Hakimi, *Optimum distribution of switching centers in a communication network and some related graph theoretic problems*, Operations Research, 13 (1965), pp. 462–475.

[79] P. Hansen, J. Brimberg, D. Urošević, and N. Mladenović, *Solving large p-median clustering problems by primal–dual variable neighborhood search*, Data Mining and Knowledge Discovery, 19 (2009), pp. 351–375.

[80] I. V. Hicks, *Branchwidth heuristics*, Congressus Numerantium, 159 (2002), pp. 31–50.

[81] I. V. Hicks, *Branch decompositions and minor containment*, Networks, 43 (2004), pp. 1–9.

[82] I. V. Hicks, *Planar branch decompositions i: The ratcatcher*, INFORMS Journal on Computing, 17 (2005), pp. 402–412.

[83] L. Hogben, M. Huynh, N. Kingsley, S. Meyer, S. Walker, and M. Young, *Propagation time for zero forcing on a graph*, Discrete Applied Mathematics, 160 (2012), pp. 1994 – 2005.

[84] C. M. Hosage and M. F. Goodchild, *Discrete space location-allocation solutions from genetic algorithms*, Annals of Operations Research, 6 (1986), pp. 35–46.

[85] M. Hribar and M. Daskin, *A dynamic programming heuristic for the p-median problem*, European Journal of Operational Research, 101 (1997), pp. 499 – 508.

[86] L.-H. Huang, G. J. Chang, and H.-G. Yeh, *On minimum rank and zero forcing sets of a graph*, Linear Algebra and its Applications, 432 (2010), pp. 2961 – 2973.

[87] C. X. Kang and E. Yi, *Probabalistic zero forcing in graphs*, Bulletin of the Institute of Combinatorial Mathematics and its Applications, 67 (2013), pp. 9–16.

[88] O. Kariv and S. L. Hakimi, *An algorithmic approach to network location problems. ii: The p-medians*, SIAM Journal on Applied Mathematics, 37 (1979), pp. 539–560.

[89] N. Karmarkar, *A new polynomial-time algorithm for linear programming*, in Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, STOC '84, New York, NY, USA, 1984, ACM, pp. 302–311.

[90] L. Khachiyan, *Polynomial algorithms in linear programming*, USSR Computational Mathematics and Mathematical Physics, 20 (1980), pp. 53 – 72.

[91] A. A. Kuehn and M. J. Hamburger, *A heuristic program for locating warehouses*, Management Science, 9 (1963), pp. 643–666.

[92] A. Kunkel, E. Van Itallie, and D. Wu, *Optimal distribution of medical backpacks and health surveillance assistants in malawi*, Health Care Management Science, 17 (2014), pp. 230–244.

[93] S. Li and O. Svensson, *Approximating k-median via pseudo-approximation*, SIAM Journal on Computing, 45 (2016), pp. 530–547.

[94] Y.-Y. Liu, J.-J. Slotine, and A.-L. Barabási, *Controllability of complex networks*, Nature, 473, pp. 167–173.

[95] F. E. Maranzana, *On the location of supply points to minimize transport costs*, Operational Research Quarterly, 15 (1964), pp. 261–270.

[96] F. Margot, *Symmetry in integer linear programming*, in 50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art,

M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 647–686.

[97] S. A. Meyer, *Zero forcing sets and bipartite circulants*, Linear Algebra and its Applications, 436 (2012), pp. 888 – 900.

[98] C. E. Miller, A. W. Tucker, and R. A. Zemlin, *Integer programming formulation of traveling salesman problems*, Journal of the ACM, 7 (1960), pp. 326–329.

[99] E. Minieka, *Conditional centers and medians of a graph*, Networks, 10 (1980), pp. 265–272.

[100] N. Mladenović, J. Brimberg, P. Hansen, and J. A. Moreno-Pérez, *The p-median problem: A survey of metaheuristic approaches*, European Journal of Operational Research, 179 (2007), pp. 927 – 939.

[101] N. Monshizadeh, S. Zhang, and M. K. Camlibel, *Zero forcing sets and controllability of dynamical systems defined on graphs*, IEEE Transactions on Automatic Control, 59 (2014), pp. 2562–2567.

[102] J. M. Mulvey and H. P. Crowder, *Cluster analysis: An application of lagrangian relaxation*, Management Science, 25 (1979), pp. 329–340.

[103] S. C. Narula, U. I. Ogbu, and H. M. Samuelsson, *Technical note—An algorithm for the p-median problem*, Operations Research, 25 (1977), pp. 709–713.

[104] R. T. Ng and J. Han, *Efficient and effective clustering methods for spatial data mining*, in Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, San Francisco, CA, USA, 1994, Morgan Kaufmann Publishers Inc., pp. 144–155.

[105] M. Pérez, F. Almeida, and J. M. Moreno-Vega, *A hybrid grasp-path relinking algorithm for the capacitated p–hub median problem*, in Hybrid Metaheuristics: Second International Workshop, HM 2005, Barcelona, Spain, August 29-30, 2005. Proceedings, M. J. Blesa, C. Blum, A. Roli, and M. Sampels, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 142–153.

[106] Z. Puchała, *Local controllability of quantum systems*, Quantum Information Processing, 12 (2012), pp. 459–466.

[107] F. P. Quintão, A. S. da Cunha, G. R. Mateus, and A. Lucena, *The k-cardinality tree problem: Reformulations and lagrangian relaxation*, Discrete Applied Mathematics, 158 (2010), pp. 1305 – 1314. Traces from LAGOS07 {IV} Latin American Algorithms, Graphs, and Optimization Symposium Puerto Varas - 2007.

[108] P. Rebreyend, L. Lemarchand, and R. Euler, *A computational comparison of different algorithms for very large p-median problems*, in Evolutionary Computation in Combinatorial Optimization: 15th European Conference, EvoCOP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings, G. Ochoa and F. Chicano, eds., Cham, 2015, Springer International Publishing, pp. 13–24.

[109] J. Reese, *Solution methods for the p-median problem: An annotated bibliography*, Networks, 48 (2006), pp. 125–142.

[110] G. Reinelt, *TSPLIB - A Traveling Salesman Problem Library*, INFORMS Journal on Computing, 3 (1991), pp. 376–384.

[111] M. G. C. Resende and R. Werneck, *A hybrid heuristic for the p-median problem*, Journal of Heuristics, 10 (2004), pp. 59–88.

[112] C. S. ReVelle and R. W. Swain, *Central facilities location*, Geographical Analysis, 2 (1970), pp. 30–42.

[113] N. Robertson and P. D. Seymour, *Graph minors. x. obstructions to tree-decomposition*, Journal of Combinatorial Theory, Series B, 52 (1991), pp. 153 – 190.

[114] E. Rolland, D. Schilling, and J. Current, *An efficient tabu search procedure for the p-median problem*, European Journal of Operational Research, 96 (1997), pp. 329 – 342.

[115] K. Rosing and C. ReVelle, *Heuristic concentration: Two stage solution construction*, European Journal of Operational Research, 97 (1997), pp. 75 – 86.

[116] K. Rosing, C. ReVelle, E. Rolland, D. Schilling, and J. Current, *Heuristic concentration and tabu search: A head to head comparison*, European Journal of Operational Research, 104 (1998), pp. 93 – 99.

[117] K. Rosing, C. ReVelle, and D. Schilling, *A gamma heuristic for the p-median problem*, European Journal of Operational Research, 117 (1999),

pp. 522 – 532.

[118] D. D. Row, *Zero forcing number, path cover number, and maximum nullity of cacti*, Involve, 4 (2011), pp. 277–291.

[119] ——, *A technique for computing the zero forcing number of a graph with a cut-vertex*, Linear Algebra and its Applications, 436 (2012), pp. 4423 – 4432. Special Issue on Matrices Described by Patterns.

[120] J. Sáez-Aguado and P. C. Trandafir, *Some heuristic methods for solving p-median problems with a coverage constraint*, European Journal of Operational Research, 220 (2012), pp. 320 – 327.

[121] E. L. Senne and L. A. Lorena, *Stabilizing column generation using lagrangean/surrogate relaxation: an application to p-median location problems.* EURO 2001 - THE EUROPEAN OPERATIONAL RESEARCH CONFERENCE - Erasmus University Rotterdam, July 9-11, 2001.

[122] E. L. Senne, L. A. Lorena, and M. A. Pereira, *A branch-and-price approach to p-median location problems*, Computers & Operations Research, 32 (2005), pp. 1655 – 1664.

[123] D. Serra, C. ReVelle, and K. Rosing, *Surviving in a competitive spatial market: The threshold capture model*, Journal of Regional Science, 39 (1999), pp. 637–650.

[124] S. Severini, *Nondiscriminatory propagation on trees*, Journal of Physics A: Mathematical and Theoretical, 41 (2008), p. 482002.

[125] P. D. Seymour and R. Thomas, *Call routing and the ratcatcher*, Combinatorica, 14 (1994), pp. 217–241.

[126] F. A. Taklimi, S. Fallat, and K. Meagher, *On the relationships between zero forcing numbers and certain graph coverings*, Special Matrices, 2 (2014), pp. 30–45, electronic only.

[127] A. Tamir, *Obnoxious facility location on graphs*, SIAM Journal on Discrete Mathematics, 4 (1991), pp. 550–567.

[128] M. B. Teitz and P. Bart, *Heuristic methods for estimating the generalized vertex median of a weighted graph*, Operations Research, 16 (1968), pp. 955–961.

[129] The Sage Developers, *SageMath, the Sage Mathematics Software System*, 2016. `http://www.sagemath.org`.

[130] M. Trefois and J.-C. Delvenne, *Zero forcing number, constrained matchings and strong structural controllability*, Linear Algebra and its Applications, 484 (2015), pp. 199 – 218.

[131] Y. Wang, A. Buchanan, and S. Butenko, *On imposing connectivity constraints in integer programs*, Mathematical Programming, (2017), pp. 1–31.

[132] N. Warnberg, *Positive semidefinite propagation time*, Discrete Applied Mathematics, 198 (2016), pp. 274 – 290.

[133] D. J. Watts and S. H. Strogatz, *Collective dynamics of 'small-world' networks*, Nature, 393 (1998), pp. 440–442.

[134] S. Welch and S. Salhi, *The obnoxious p facility network location problem with facility interaction*, European Journal of Operational Research, 102 (1997), pp. 302 – 319.

[135] E. Yi, *On zero forcing number of permutation graphs*, in Combinatorial Optimization and Applications: 6th International Conference, COCOA 2012, Banff, AB, Canada, August 5-9, 2012. Proceedings, G. Lin, ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 61–72.