ADVANCEMENTS ON PROBLEMS INVOLVING MAXIMUM FLOWS

A Thesis
Presented to
The Academic Faculty

by

Douglas S. Altner

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Industrial and Systems Engineering

Georgia Institute of Technology
August 2008

ADVANCEMENTS ON PROBLEMS INVOLVING MAXIMUM FLOWS

To my parents.

ACKNOWLEDGEMENTS

I have the sincerest appreciation for my thesis advisor, Dr. Özlem Ergun. Her continuous guidance, encouragement, patience and financial support was instrumental in making the research contained within possible as well as in allowing me to develop as a researcher. Words cannot express the full extent of my gratitude towards her valuable influence as my advisor and a role model.

I thank Dr. Shabbir Ahmed, Dr. William Cook, Dr. Dana Randall and Dr. Joel Sokol for serving on my dissertation committee and for their valuable feedback. I am proud to have the opportunity to present my research to such accomplished academics. In addition, I thank Dr. Ravindra Ahuja and his research group for allowing me to collaborate with them on railroad optimization problems in the summer of 2007. Although brief, this experience has been very influential on my future research interests. Furthermore, I also thank Dr. Faiz Al-Khayyal for helping me develop as a teacher.

I am also very grateful towards many of my fellow doctoral students at Georgia Tech, most notably Alejandro, Andrew, Chris, Dan, Fatma, Fred, Guanghui, Juan Pablo, Kael and Ricardo as well as Nelson from MIT's Operations Research Center. All of the aforementioned have supported me on both technical matters and in friendship. Although there are too many to name explicitly, there are also many others who have still earned my sincere appreciation and they all should rest assured that they are appreciated.

Finally, I would like to thank my parents, my brother and my grandmother for being supportive throughout all of my years of schooling.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis presents both theoretical and computational advancements on problems involving maximum flows. Four problems are explored: the Maximum Flow Network Interdiction Problem (MFNIP), the Maximum Flow Single Arc Reoptimization Problem (MFSAROP), The Robust Minimum Capacity s-t Cut Problem (RobuCut) and the Two-Stage Stochastic Maximum Flow Problem (2SMFP). Discussions and analyses of each of these four problems compose chapters 2-6 of this thesis. The first chapter is an introduction to network programming in general, with an emphasis on the Maximum Flow Problem.

The first problem discussed is MFNIP. Although there have been papers on various optimization models of similar network interdiction problems since the 1950s (e.g., [41]), MFNIP, in particular, is first introduced in Wood [77]. MFNIP is defined as the following problem: given a capacitated s-t flow network, allocate a finite amount of resources to delete arcs from the network so as to minimize the maximum flow in the network induced on the set of undeleted arcs. This problem has numerous nontrivial applications including those in intercepting smuggled nuclear material [23], combatting illegal drug trafficking [77], sewage treatment [63] and maintaining infection control in a hospital [7].

Wood proved this problem is strongly NP-hard [77] and there are several results on solving instances of MFNIP using Lagrangian relaxation (e.g., [11], [74] and [67].) However, very few theoretical results on the widely used integer linear programming (ILP) formulation of Wood [77] are documented. In the second chapter of this thesis, we present several new results on the Cardinality Maximum Flow Network Interdiction Problem (CMFNIP), which is the special case of MFNIP where the interdictor removes a fixed number of arcs from the network. Studying CMFNIP is important; it is also known as the n Most Vital Arcs Problem and it has nontrivial applications in flood control [65].

With regards to new theoretical results, first, we introduce two exponentially large classes of valid inequalities for CMFNIP and prove that they can be separated in polynomial time. Second, we prove a non-constant lower bound on the integrality gap of Wood's ILP and show that this same bound holds even when the LP relaxation is strengthened with our two classes of valid inequalities. Both the introduction of polynomial-time separable, exponentially large valid inequalities for Wood's ILP and the lower bound on the integrality gap of Wood's ILP are the first known results of their kind. We note that one of our classes of valid inequalities is a natural extension of the Type I inequalities introduced by Wood [77].

In the third chapter, we illustrate that the time required for CPLEX to solve an instance of MFNIP to optimality is extremely sensitive on the interdiction budget. As an alternate solution approach to MFNIP, we design a variable-depth flip neighborhood as well as a meta-heuristic framework to allow escape from locally optimal solutions. We demonstrate that our neighborhood terminates with good quality locally optimal solutions in an amount of time that demonstrates to be relatively insensitive to the value of interdiction budget.

The fourth chapter of this thesis concerns rapidly solving an online sequence of Maximum Flow Problems (MFPs) on topologically similar networks. There are a diverse collection of applications that might require one to solve such a sequence of MFPs. This includes, but is not limited to, estimating the physical difference between the tertiary structures of two proteins [72], searching a database of fingerprints [40]

and computing an expected maximum flow in the context of two-stage stochastic network programming [75] and [76].

To initiate the study of solving an online sequence of MFPs, we examine a problem where the networks of every two adjacent MFPs in the sequence differ from each other by exactly one arc; all other arcs possess the same capacity. We hereby refer to this special case as the Maximum Flow Single Arc Reoptimization Problem (MFSAROP) and we note that it has applications in scheduling jobs on a dual-processor machine in real-time [71].

To efficiently solve an entire online sequence of MFPs, we modify the highly regarded pre-flow push algorithm of Goldberg and Tarjan [38] so that it is possible to warm start the algorithm from solutions that do not satisfy the flow balance constraints. We observe that when a single arc is added or deleted in a MFP that was solved to optimality, there are fundamentally four possible cases for reoptimization. To exploit this fact, we store two minimum *s*-*t* cuts from the previous MFP, in addition to the optimal residual network, to heuristically identify which of the four fundamental reoptimization cases modifying a given arc leads to. We demonstrate that our algorithm is very practical as it typically uses around 15% of the time required to solve an instance of MFSAROP compared to an approach that repeatedly uses a black-box maximum flow solver to solve each MFP.

In the fifth chapter of this thesis, we develop maximum flow reoptimization heuristics for computing a robust minimum cut under the Bertsimas and Sim model of robustness [10]. We hereby refer to this problem as the Robust Minimum Capacity *s*-*t* Cut Problem (RobuCut). RobuCut models the decision of conservatively choosing an *s*-*t* cut in light of data uncertainty on the arc capacities. RobuCut extends nicely to many real-world applications of the Minimum Capacity *s*-*t* Cut Problem (MinCut) where data uncertainty on the arc capacities is a serious possibility. In particular, we speculate that RobuCut has applications in open-pit mining [43], project scheduling [55] and compiler optimization [78] and [79].

To solve RobuCut, we develop a special implementation of the algorithm of Bertsimas and Sim for robust combinatorial optimization problems (RobuCOPs). Our implementation requires that a sequence of MFPs be solved. We identify that the sequence has several properties that can be exploited for efficient maximum flow reoptimization techniques. We demonstrate that our algorithm is very fast in practice, as it can solve several RobuCut instances in under thirty seconds that would normally require over four hours if we use an approach that repeatedly uses a black-box maximum flow solver to solve each MFP.

The sixth chapter of this thesis details research on developing maximum flow reoptimization heuristics for solving what we formulate as the Two-Stage Stochastic Maximum Flow Problem (2SMFP). 2SMFP assumes we have a maximum flow network, a finite budget for increasing arc capacities and a randomly distributed *operating level* for each arc, which dictates whether the arc *fails.* That is, the arc gets removed from the network after the capacities have been installed but before flow may be sent. Assuming there are a polynomial number of possible scenarios for arc failures, the objective is to use the budget to increase arc capacities so as to maximize the expected maximum flow.

In this chapter, we present a Benders' reformulation of 2SMFP and provide a cutting plane algorithm to obtain an optimal solution. The structure of the cutting plane algorithm motivates two possible approaches for incorporating maximum flow reoptimization heuristics, both of which are implemented and tested. We conclude that our implementations can obtain very modest, but never substantial, savings on solving 2SMFP when compared to an implementation that iteratively uses a black-box maximum flow solver.

CHAPTER I

INTRODUCTION

This chapter provides a general overview of network programming. The second section presents an essentialized history of the Maximum Flow Problem. The final section of this chapter offers a summary of this thesis.

1.1 Network Optimization

This thesis concerns itself with the area of *network programming.* That is, the study of mathematical programs where a subset of the decision variables correspond to routing decisions on a network. *Network flows* typically refers to the study of algorithms to solve optimization problems on networks. Before discussing what a network is in the context of mathematics, we first highlight the importance of and motivation for studying networks. Networks abstract the essential infrastructure-related issues from human industry. For example, the electricity industry depends on electricity grids to power homes, offices, factories and warehouses. The internet, arguably the largest network created by mankind, has revolutionized the frequency, quantity and quality of information that can be shared for both business and recreational purposes. Road, rail, airline service and sea cargo networks compose the lifeblood of the global economy as they allow for the distribution of food, raw materials, health supplies and consumer products. Everywhere one looks, it is undeniable that networks are critical to the flourishing of human industry and therefore to raising standard of living and advancing civilization.

Generally speaking, a network is a discrete structure that defines connectivity relationships amongst a set of *nodes*. In this context, an *arc* denotes a direct connection between two nodes. Arcs typically have an *orientation*, which indicates which node the arc originates from and which node the arc terminates in. For example, if our network is modeling an interstate road network, then the nodes might correspond to cities and the arcs could correspond to highways. Analogously, if our network is modeling the internet, then nodes could correspond to webpages and arcs could represent hyperlinks between websites.

A *flow* on an arc denotes the number of units of the commodity in question, be it electricity, freight, fluid, et cetera, that will traverse the arc. A *network flow* is an assignment of a value of flow to each arc in the network. A network flow is *balanced* if, for each node, the total amount of the flow entering the node equals the total amount of the flow leaving the node. For many real-world applications of network optimization, the requirement that the network flow be balanced is natural and often non-negotiable.

In addition to having an orientation and a flow value, arcs typically have an associated *cost*, which often denotes the cost per unit of flow that traverses the arc. In addition, arcs have flow bounds, that respectively denote the minimum and maximum amount of flow that may traverse an arc. Often, an upper bound on the amount of flow that may traverse an arc is called an arc *capacity*.

Unless otherwise stated, it is typically assumed that a network has no parallel arcs and that no arc terminates in the same node that it has originated from. In addition, it is typically assumed that all units of flow in a network are of the same, indistinguishable commodity. Lastly, unless otherwise stated, it is often assumed that all flow in a network travels instantaneously.

In all of such applications of networks, a decision maker typically wishes to move an entity, typically merchandise, passengers, information, bulk material or the like, from one location to another as efficiently as possible. The reasoning for doing so is often to provide quality service to customers or to utilize one's resources efficiently. This motivates the study of *network optimization.* Given the ubiquity of networks in human industry and the large-scale impact that good, large-scale routing decisions can have in raising the standard of living, we see that the study of network optimization is well motivated.

Network optimization is crucial to many different academic areas, including, but not limited to, operations research, applied mathematics, computer science, hydraulics engineering, civil engineering and operations management. Arguably, the first network optimizer in history was Charles Babbage, who worked on transportation and sorting operations for England's postal system. Interestingly, Babbage's efforts played an integral part in the universal "penny post" of England during the middle of the 19th century [70]. Another significant advance in the history of network optimization came with Gustav Kirchhoff and the other electrical engineers who first systematically analyzed electrical circuits in the later part of the 19th century.

The next major milestone of network optimization was during World War II, where scientists in the United Kingdom including Patrick Blackett, Cecil Gordon, C. H. Waddington, Owen Wansbrough-Jones and Frank Yates along with United Statesman George Dantzig investigated formal methods for making intelligent logistical decisions. After the second world war the science of operations research, including network optimization, began to be applied to similar problems in industry.

There are three fundamental problems in the study of network optimization. The first is called *The Shortest Path Problem*, which can be stated as follows: given a network with a starting location, called a *source*, an ending location, called a *sink*, and an associated *length* on each arc, find the path of shortest length from the source to the sink.

The second fundamental problem of network optimization, which is also the keystone of this thesis, is called the *Maximum Flow Problem*. This problem can be stated as follows: given a network with a source, a sink and a capacity on each arc, find the maximum possible flow that can be routed through the network from the source to the sink. This problem is often used to model real-world situations where the routing costs are negligible relative to the profit per unit of flow.

The third fundamental problem of network optimization is the *Minimum Cost Flow Problem*. This problem can be stated as follows: given a network where each arc has an associated cost per unit of flow, a lower bound and an upper bound on the flow that may traverse the arc and each node either has an associated *supply*, an amount of flow that must originate from the node or an associated *demand*, the amount of flow that must terminate in the node or is a *transhipment node*, that is, has no associated supply or demand, find a minimum cost flow in the network such that all demands are satisfied. Both the Shortest Path Problem and the Maximum Flow Problem may be modeled as Minimum Cost Flow Problems.

Although there are numerous other widely studied optimization problems that occur on networks, such as multicommodity flow problems, the Traveling Salesman Problem and vehicle routing problems, we omit them from this discussion. These problems are very difficult to solve in practice, and are not approached with network flow algorithms. Usually these problems are approached with a combination of integer programming techniques as well as optimization heuristics.

1.2 The Maximum Flow Problem

This thesis focuses on several problems related to the Maximum Flow Problem (MFP). MFP in and of itself has numerous applications in mathematical modeling, including those in internet traffic routing [3], railroad freight transportation [2], trucking [3], open-pit mining [43], sports team elimination [22], airline scheduling [3] and numerical linear algebra [3].

There are many interesting properties of the MFP. First of all, there is the property of *integrality.* If the capacities on all of the arcs in the network are integers, then there will be a maximum flow in the network such that every arc will assume an integer value and the total flow through the network will be integer. This property is important because integrality constraints are typically very difficult to enforce in practice, which is why integer linear programs are much more difficult to solve in practice than linear programs. The integrality property of maximum flows allows the integrality on the arc flows to be guaranteed by merely assigning integer arc capacities. Thus, a maximum flow problem can be solved as a linear program, despite integrality requirements on the flow.

A second important property of MFP is known as the Maximum Flow Minimum Cut Theorem. Before we describe the implication of this theorem, first we introduce a few concepts. Given a network N with a source s and a sink t, a *s-t cut* is a set of arcs C such that every path from s to t contains at least one arc in C. The *capacity* of a s-t cut is the sum of all of the capacities of the arcs in the cut. The Maximum Flow Minimum Cut Theorem states that the value of the maximum flow from s to t equals the capacity of the minimum capacity s-t cut. This property is very intuitive. In the context of modeling a series of pipes, this theorem is conceptually equivalent to stating that the maximum flow through a system is limited by the system's bottleneck.

Ford and Fulkerson presented the first formal treatment of the Maximum Flow Problem in [30]. Ford and Fulkerson were also the first to identify the Maximum Flow Minimum Cut Theorem. As a side note, the Minimum Capacity *s*-*t* Cut Problem first appeared in [41], where researchers at RAND Corporation modeled the throughput of the Soviet railroad network during the Cold War.

In [30], Ford and Fulkerson also present an algorithm for computing a maximum flow in a network. Their algorithm, although ground breaking in its time, is conceptually simple. The fundamental idea is to maintain a *residual network*, that is, a network that allows the user to either increase the flow on an arc with unused capacity or to decrease the flow that has already been tentatively routed on an arc. The algorithm iteratively augments flows along a path in the residual network until no more such paths exist.

So long as the maximum flow in the network is finite, the algorithm of Ford and Fulkerson is guaranteed to terminate in a finite number of iterations with a maximum flow. However, there is no guarantee that the Ford-Fulkerson algorithm will terminate in a small number of iterations with respect to the number of nodes and the number of arcs in the network. In fact, it is possible for the algorithm to require on the order of c_{max} iterations, where c_{max} denotes the largest arc capacity in the network.

After Ford and Fulkerson, there has been an intensive study of algorithms for MFP. For a nice survey on worst-case complexity results, please see [35]. Important milestones include the algorithm of Edmonds and Karp [29], which is an implementation of the Ford-Fulkerson Algorithm that iteratively finds the *shortest* augmenting path and terminates with a polynomial number of computations with respect to the problem input. In [27], Dinic presents the first maximum flow algorithm based on the idea of iteratively solving for maximal flows in a *layered network*. In [47], Karzanov presents the first algorithm using the concept of *pre-flows*, that is, an assignment of flow values to the arcs where the amount of flow entering a node is greater than or equal to the amount of flow leaving a node. Karzanov's algorithm combines the use of pre-flows with layered networks to develop a new maximum flow algorithm. In [52], Malhotra et al. introduce the eponymous MKM Algorithm, a maximum flow algorithm that is based on iteratively assigning flow on arcs incident to node with the smallest throughput in the residual network.

The science of computing maximum flows drastically changed after the publication of Goldberg and Tarjan [38]. Goldberg and Tarjan present a widely implemented algorithm that uses pre-flows in conjunction with a new concept called *distance labels*, which are labels on the nodes that are a lower bound on the length of the shortest path, in terms of the number of arcs, from the node to the sink in the residual network. In [4], Ahuja and Orlin introduce a maximum flow algorithm that is based on the idea of *capacity scaling.* That is, iteratively finding paths with relatively large throughput capability. In [37], Goldberg and Rao introduce an innovative concept of a binary arc length function. In many previous maximum flow algorithms, such as the Goldberg-Tarjan algorithm [38], there is a concept of distance that assumes all arcs have a length of 1 unit. In order to obtain a better worst-case algorithmic result, Goldberg and Rao relax this assumption by allowing arc lengths to assume values 0 or 1. This algorithm has receives a lot of attention because it achieves the best known worst-case algorithmic result for MFP: $O(min(|V|^{\frac{2}{3}}, \sqrt{|A|})\ |A|\ log(\frac{|V|^2}{|A|})\ log(c_{max}))$ where $c_{max} = max\{c_e | e \in A\}$. This result is for a MFP on a network $N = (V, A)$ where c_e denotes the capacity of arc e.

One final algorithm that is worthy of note is the algorithm of Fujishige [32]. At each iteration, this algorithm constructs an augmenting subgraph using *maximum adjacency orderings*, which are a concept that is introduced in [56]. This algorithm is interesting in that at each iteration, it can augment more flow than can be augmented on any single augmenting path in the present residual network. In [53], Matsuoka and Fujishige test an improved version of Fujishige's algorithm that uses pre-flows and demonstrate that it outperforms the Goldberg-Tarjan on a few classes of randomly generated instances of MFP.

Despite a flurry algorithmic ideas for the maximum flow problem, the algorithm of Goldberg and Tarjan is still considered the most efficient in practice [19] and [53].

CHAPTER II

CARDINALITY MAXIMUM FLOW NETWORK INTERDICTION

This chapter discusses theoretical results on the Cardinality Maximum Flow Network Interdiction Problem (CMFNIP). The first section introduces CMFNIP and briefly overviews relevant literature. In the second section we discuss previous results on CMFNIP, including the widely used ILP of Wood [77]. In the third section, we introduce our two exponentially large classes of valid inequalities for Wood's ILP for CMFNIP and prove that they can be separated in polynomial-time. In the fourth section, we prove that the integrality gap for Wood's ILP is contained in the set $\Omega(|V|^{1-\epsilon})$ for any $\epsilon \in (0,1)$, even when the LP relaxation is strengthened with our polynomially-separable valid inequalities. Here, $|V|$ denotes the number of nodes in the underlying network. In the fifth section, we discuss the hardness of approximation implication of the above integrality gap result and provide an approximating-factor-preserving reduction from an interdiction problem that is much simpler in structure to the more general Maximum Flow Network Interdiction Problem (MFNIP). In the final section we draw conclusions and state an open problem concerning the approximability of MFNIP.

2.1 *Introduction*

The Maximum Flow Network Interdiction Problem (MFNIP) is defined as a Stackelberg game where an *interdictor* allocates a finite amount of resources towards removing arcs from a network so as to minimize the maximum flow that can be routed through the remaining network. Since this is a Stackelberg game, the arcs are first removed from the network before the *adversary* routes his flow.

Interdiction problems abstract the essential issues in many real-world resource allocation problems including, but not limited to, military applications [54], combatting drug trafficking [77], controlling the spread of disease in a hospital [7], chemically treating raw sewage [63] and controlling floods in a system of dams or in a sewage system [65].

Network interdiction is important in the history of operations research. An interdiction problem motivated the first application of the Minimum Capacity s-t Cut Problem. During the Cold War, analysts at the RAND Corporation were researching how to interdict the Soviet Union's railroad traffic into Eastern Europe using the fewest possible interdiction resources. To do this, requires the computation of a minimum capacity s-t cut and hence is the earliest known formulation of this fundamental problem [41].

From the 1960's to the turn of the 21st century there has been an extensive amount of academic literature on various interdiction problems, most of which is listed in [21]. The basic framework for maximum flow network interdiction when the interdictor could not afford to stop all traffic was first studied, with minor variants, in [54]. In the early 1990's, Wood resurrected interest in MFNIP with the widely cited [77]. A relaxation of MFNIP, called the Network Inhibition Problem (NIP), has also been independently explored by Phillips [63].

A broad class of network interdiction problems have been intensively studied. This includes, but is not limited to, shortest path network interdiction [44], stochastic network interdiction [23], [45], multiple commodity network interdiction [51], [77], facility interdiction [21] and a variant where the adversary routes flow before arcs are removed [28]. There is also literature on more-than-two-stage interdiction models where infrastructure may be reinforced against attacks [13].

Essentially all of the recent work on MFNIP (e.g., [11], [67], [74] and [77]) uses an integer linear programming formulation (ILP) that was originally introduced by Wood in [77], which we hereby refer to as Wood's ILP. Wood's ILP is also used as a starting point for the mathematical programming formulations of many of the extensions of MFNIP (e.g., [23], [45] and [51]).

Of particular interest is the special case of MFNIP when an interdictor removes exactly R arcs from the network in order to minimize the maximum flow in the resulting network. This is known as the Cardinality Maximum Flow Network Interdiction Problem (CMFNIP) and is equivalent to the R-Most Vital Arcs Problem, which is cited to have applications in controlling floods in a system of dams or a sewage system as well as military applications [65].

Despite its wide use in nearly all papers on maximum flow network interdiction, there are not many theoretical results on Wood's ILP. The first main contribution of this chapter is to introduce two new exponentially large classes of valid inequalities for Wood's ILP for CMFNIP along with polynomial-time separation algorithms for each. The first of the two classes introduced is a generalization of the "Type I" valid inequalities provided by Wood [77]. These are the first documented separation results for valid inequalities for CMFNIP.

The second major contribution of this chapter is that we identify and prove that the integrality gap of Wood's ILP for CMFNIP is contained in the set $\Omega(|V|^{1-\epsilon})$ even when the LP relaxation of Wood's ILP is strengthened with our two classes of valid inequalities. In this context, $|V|$ is the number of nodes in the network and ϵ is any constant in $(0, 1)$. In other words, we prove that there is no constant factor lower bound to the integrality gap. Note that this result also applies for Wood's ILP for CMFNIP, since this would be a relaxation of the ILP which we studied, as well as

Wood's ILP for general MFNIP.

Bounds on integrality gaps are commonly sought theoretical results in integer linear programming. The integrality gap is a commonly used metric to assess the quality of a linear programming (LP) relaxation, which are commonly used as part of implicit enumeration techniques in commercial integer programming solvers. In addition, knowing the integrality gap of an ILP often provides general problem insight, bounds for LP-based approximation algorithms and a method for evaluating the quality of heuristically obtained solutions.

This chapter also contains contributions on the hardness of approximating MFNIP. An immediate corollary of our integrality gap result is that it is NP-hard to obtain a $O(|V|^{1-\epsilon})$-approximate optimal solution to MFNIP when using the LP relaxation of Wood's ILP as a lower bound. We are currently unable to resolve the question if this result is true for any arbitrary lower bound. However, we do show that any hardness of approximation result on the R-Interdiction Covering Problem (RIC), which is a facility interdiction version of MFNIP that is much simpler in structure but still strongly NP-hard, immediately extends to MFNIP. Thus, future researchers may focus on a problem of simpler structure, RIC, to indirectly obtain a hardness of approximation result on MFNIP. Hardness of approximation results are often insightful because they detail the worst-case complexity of obtaining a polynomial-time algorithm that guarantees a solution within a provable factor of the optimal solution for a given optimization problem.

With regards to related literature, there have been a few approximation algorithm results on NIP, which is a relaxation of MFNIP where arc interdictions may be continuous instead of binary. In [63], Phillips introduced three pseudopolynomial-time algorithms for the special case of NIP on planar networks as well as instructions for extending these algorithms into fully-polynomial-time approximation schemes (FPTAS). In [14], Burch et al. provide a polynomial-time algorithm for NIP that either returns a $(1+\frac{1}{\epsilon})$-approximate optimal solution or a $(1+\epsilon)$-pseudoapproximation. However, it is not known a priori which solution is returned. In this context, ϵ is a user-specified error parameter.

2.2 Wood's Formulation of MFNIP

Wood's ILP for MFNIP can be viewed as a Stackelberg game. First the interdictor chooses what arcs to interdict. After the arcs have been removed, the adversary, with perfect information on which arcs have been removed, chooses a minimum capacity s-t cut in the network induced on the set of non-interdicted arcs.

We denote the value of the interdiction budget by R. In addition, for any arc e we denote its flow capacity by c_e and its interdiction cost by r_e. All data for this problem are positive integers.

Decision Variables

$$\alpha_i := \begin{cases} 1 & \text{if } i \in N \text{ is on the sink side of the cut.} \\ 0 & \text{otherwise.} \end{cases}$$

$$\beta_e := \begin{cases} 1 & \text{if } e \in A \text{ is in the cut and is interdicted.} \\ 0 & \text{otherwise.} \end{cases}$$

$$\gamma_e := \begin{cases} 1 & \text{if } e \in A \text{ is in the cut and is not interdicted.} \\ 0 & \text{otherwise.} \end{cases}$$

Formulation

Minimize $\quad \sum_{e\in A} c_e\gamma_e$

Subject to

$$\alpha_u - \alpha_v + \beta_{(u,v)} + \gamma_{(u,v)} \geq 0 \quad \forall (u,v) \in A$$

$$\alpha_t - \alpha_s \geq 1$$

$$\sum_{e\in A} r_e\beta_e \leq R$$

$$\begin{array}{ll} \alpha_i \in \{0,1\} & \forall i \in N \\ \beta_e \in \{0,1\} & \forall e \in A \\ \gamma_e \in \{0,1\} & \forall e \in A \end{array}$$

The first set of constraints enforce that if arc (u, v) is in the cut defined by the α variables, then (u, v) is either interdicted or is not interdiction. One such if-then constraint exists for each arc (u, v).

The second line of constraints fixes α_t to 1 and α_s to 0.

The third line of constraints enforces that the interdiction budget is not exceeded. We refer to this as the *knapsack constraint.*

To use Wood's ILP to model CMFNIP, set $r_e = 1$ for each arc $e \in A$.

Theorem 1. *CMFNIP (and therefore MFNIP) is strongly NP-hard.*

Proof: See [77]. The argument is a reduction from the Maximum Clique Problem. □

We define the *linear programming relaxation of Wood's formulation* to be identical to the ILP but where the binary constraint on each variable is replaced with a corresponding lower bound of 0 and an upper bound of 1 on that same variable.

We note that we have reversed the roles of the β and the γ decision variables from Wood's paper. In [77], Wood uses γ as an indicator decision variable to denote if an arc is in the s-t cut defined by the α variables that is also interdicted and uses β as an indicator decision variable to denote if an arc is in the s-t cut defined by the α variables that is not interdicted.

2.3 Strengthening the LP Relaxation of Wood's ILP for CMFNIP

In this section, we present two general, polynomial-time separable classes of valid inequalities for CMFNIP. In the first subsection, we introduce our *node-to-sink path valid inequalities*, prove their validity and provide a polynomial-time separation algorithm. In the second subsection, we do the same for another class of valid inequalities that we call the *source-to-node path valid inequalities.*

2.3.1 Node-to-Sink Path Valid Inequalities

Consider an arbitrary instance of CMFNIP on a network $N = (V, A)$ with interdiction budget R. For any node $u \in V$ on the source-side of the s-t cut defined by the α variables and let P_{u-t} be a set of arc-disjoint u-t paths in N such that $|P_{u-t}| > R$. Then, since at most R paths may be interdicted, we know that for all of the arcs in a path in P_{u-t}, at least $|P_{u-t}| - R$ of these arcs must have their corresponding γ variable equated to 1.

We can exploit this idea to develop the class of node-to-sink valid inequalities:

$$(|P_{u-t}| - R)\alpha_u + \sum_{e \in A(P_{u-t})} \gamma_e \geq |P_{u-t}| - R \qquad \forall\ u \in V;\ \forall\ P_{u-t} \in \mathcal{P}^R_{u-t} \qquad (1)$$

where $A(P_{u-t})$ denotes the set of all arcs contained in a u-t path in P_{u-t} and $\mathcal{P}^R_{u-t}$

denotes the family of all possible sets of arc-disjoint u-t paths that have cardinality strictly greater than R.

We note that this class of valid inequalities is more general than the "Type I inequalities" presented in Wood [77]. Specifically, his result is for any set of arc-disjoint u-t paths of *maximum cardinality* and of size at least $R + 1$. However, we note that the fact that the set must be maximum is inessential to the coefficient strengthening argument used in his proof of validity.

Theorem 2. *The node-to-sink inequalities are valid for Wood's ILP for CMFNIP.*

Proof: We present a combinatorial argument. Consider any node $u \in V$ and any set of paths $P_{u-t} \in \mathcal{P}^R_{u-t}$. Let $\hat{x} = (\hat{\alpha}\ \hat{\beta}\ \hat{\gamma})$ be a feasible solution to Wood's ILP for CMFNIP. If $\hat{\alpha}_u = 1$, then the inequality in Equation 1 is trivially satisfied by $\hat{x}$.

Suppose $\hat{\alpha}_u = 0$. Then, since $\hat{x}$ is a feasible solution, we know that for each u-t path $p_{u-t} \in P_{u-t}$ there exists at least one arc $e \in A(p_{u-t})$ such that $\hat{\beta}_e + \hat{\gamma}_e \geq 1$, which implies that $\sum_{e \in A(P_{u-t})} \hat{\beta}_e + \hat{\gamma}_e \geq |P_{u-t}|$. We note that the knapsack constraint implies that $\sum_{e \in A(P_{u-t})} \hat{\beta}_e \leq R$. Thus, we may conclude that $\sum_{e \in A(P_{u-t})} \hat{\gamma}_e \geq |P_{u-t}| - R$, which completes the proof. $\square$

Theorem 3. *Given an instance of CMFNIP on a network $N = (V, A)$ with interdiction budget R and a feasible solution $\hat{x} = (\hat{\alpha}\ \hat{\beta}\ \hat{\gamma})$ to the corresponding linear programming relaxation, there exists a polynomial-time algorithm that either (i) asserts that $\hat{x}$ satisfies all of the valid inequalities in Equation* 1 *or (ii) produces a violated inequality from Equation* 1.

Proof: For a node $u \in V$, we first provide a polynomial-time algorithm to separate $\hat{x}$ over all inequalities in Equation 1. This is done by solving a minimum cost flow problem. Thus, we then separate over all of the node-to-sink inequalities using $O(|V|)$ minimum cost flow computations, which achieves the desired result.

First, we provide intuition to motivate the minimum-cost-flow-based separation routine. Let $u \in V$ and let $P_{u-t} \in \mathcal{P}^R_{u-t}$. Given $\hat{x}$, we rearrange the corresponding sub-class of node-to-sink inequalities to read as follows:

$$(\hat{\alpha}_u - 1)|P_{u-t}| + \sum_{e \in A(P_{u-t})} \hat{\gamma}_e \geq (\hat{\alpha}_u - 1)R \tag{2}$$

We want to search over all sets of arc-disjoint u-t paths $P_{u-t} \in \mathcal{P}^R_{u-t}$ to find a set that minimizes the left-hand side of the inequality expressed in Equation 2. To do this, we can solve a minimum cost flow problem where each arc e in the original network has a cost of $\hat{\gamma}_e$ and each u-t path costs $(\hat{\alpha}_u - 1)$, which is non-positive.

Construct an auxiliary network $N^a = (V^a, A^a)$ as follows. For each node $v \in V$ we create a corresponding node $v^a \in V^a$. Similarly, for each arc $(i,j) \in A$ we create a corresponding arc $(i^a, j^a) \in A^a$ with a capacity of one unit and a cost per unit flow of $\hat{\gamma}_{(i,j)}$. In addition, we create another arc (t^a, u^a) with a lower bound of $R + 1$ units, an infinite capacity and a per-unit of flow cost of $(\hat{\alpha}_u - 1)$. Note that there is a bijection between every set of paths $P_{u-t} \in \mathcal{P}^R_{u-t}$ to the basic feasible solutions of the minimum cost flow problem on N^a. Moreover, note that the cost of the flow in N^a equals the left-hand side of the inequality expressed in Equation 2.

The unit capacity on each arc in $A^a \backslash \{(t^a, u^a)\}$ ensures that the corresponding u-t paths are indeed arc-disjoint. The lower bound on arc (t^a, u^a) ensures that the set of u-t paths obtained from solving the minimum cost flow problem is of cardinality strictly greater than R. If there is no such set of paths, then the minimum cost flow problem is infeasible.

We may assume without loss of generality that the optimal solution contains no zero-cost, non-zero flow circulations. It is well known that a circulation on a network with m arcs can be decomposed into a cycle flow along at most m directed cycles in

$O(m^2)$ time. See section 3.5 in Ahuja et al. [3].

If the optimal objective value of the minimum cost flow problem on network N^a is strictly less than $R(\hat{\alpha}_u - 1)$, then there is a violated node-to-sink inequality and a corresponding optimal basic feasible solution gives a set of arc-disjoint paths, which, along with u, provide the violated inequality. Similarly, if the optimal objective value of the minimum cost flow problem on network N^a is greater than or equal to $R(\hat{\alpha}_u - 1)$, then all inequalities from Equation 1 are satisfied.

Thus, if we repeat this procedure for each node $u \in V$, we obtain a polynomial-time separation routine. □

2.3.2 Source-to-Node Path Valid Inequalities

Source-to-node inequalities are very similar in spirit to node-to-sink inequalities. As before, we first make a structural insight. Consider an arbitrary instance of CMFNIP on a network $N = (V, A)$ with interdiction budget R. For any node $u \in V$ on the sink side of the s-t cut defined by the α variables and let P_{s-u} be a set of arc-disjoint s-u paths in N such that $|P_{s-u}| > R$. Then, since at most R paths may be interdicted, we know that for all of the arcs in a path in P_{s-u}, at least $|P_{s-u}| - R$ of these arcs must have their corresponding γ variable equated to 1.

Thus, employing this idea, we can create the *source-to-node valid inequalities*:

$$(R - |P_{s-u}|)\alpha_u + \sum_{e \in A(P_{s-u})} \gamma_e \geq 0 \qquad \forall\ u \in V;\ \forall\ P_{s-u} \in \mathcal{P}^R_{s-u} \tag{3}$$

where $A(P_{s-u})$ denotes the set of all arcs contained in a s-u path in P_{s-u} and $\mathcal{P}^R_{s-u}$ denotes the family of all possible sets of arc-disjoint s-u paths that have cardinality strictly greater than R.

Theorem 4. *The source-to-node inequalities are valid for Wood's ILP for CMFNIP.*

Proof: As before, we present a combinatorial argument. Consider any node $u \in V$ and any set of paths $P_{s-u} \in \mathcal{P}^R_{s-u}$. Let $\hat{x} = (\hat{\alpha}\ \hat{\beta}\ \hat{\gamma})$ be a feasible solution to Wood's ILP for CMFNIP. If $\hat{\alpha}_u = 0$, then the inequality in Equation 3 is trivially satisfied by $\hat{x}$.

Suppose $\hat{\alpha}_u = 1$. Then, since $\hat{x}$ is a feasible solution, we know that for each s-u path $p_{s-u} \in P_{s-u}$ there exists at least one arc $e \in A(p_{s-u})$ such that $\hat{\beta}_e + \hat{\gamma}_e \geq 1$, which implies that $\sum_{e \in A(P_{s-u})} \hat{\beta}_e + \hat{\gamma}_e \geq |P_{s-u}|$. We note that the knapsack constraint implies that $\sum_{e \in A(P_{s-u})} \hat{\beta}_e \leq R$. Thus, we may conclude that $R - |P_{s-u}| + \sum_{e \in A(P_{s-u})} \hat{\gamma}_e \geq 0$, which completes the proof. □

Theorem 5. *Given an an instance of CMFNIP on a network $N = (V, A)$ with interdiction budget R and a feasible solution $\hat{x} = (\hat{\alpha}\ \hat{\beta}\ \hat{\gamma})$ to the corresponding linear programming relaxation, there exists a polynomial-time algorithm that either (i) asserts that $\hat{x}$ satisfies all of the valid inequalities in Equation* 3 *or (ii) produces a violated inequality from Equation* 3*.*

Proof: As in the case of the node-to-sink valid inequalities, for a node $u \in V$, we first provide a polynomial-time algorithm to separate $\hat{x}$ over all inequalities in Equation 3. This is done by solving a minimum cost flow problem. Thus, we then separate over all of the source-to-node inequalities using $O(|V|)$ minimum cost flow computations. Since the minimum cost flow problem can be solved in polynomial-time (see Chapter 10 of Ahuja et al. [3]), this achieves the desired result.

As before, we provide intuition to motivate the minimum-cost-flow-based separation routine. Let $u \in V$ and let $P_{s-u} \in \mathcal{P}^R_{s-u}$. Given $\hat{x}$, we can rearrange the corresponding sub-class of source-to-node inequalities to read as follows:

$$-\hat{\alpha}_u |P_{s-u}| + \sum_{e \in A(P_{s-u})} \hat{\gamma}_e \geq -\hat{\alpha}_u R \tag{4}$$

We want to search over all sets of arc-disjoint u-t paths $P_{u-t} \in \mathcal{P}^R_{u-t}$ to find a set that minimizes the left-hand side of the inequality expressed in Equation 4. To do this, we can solve a minimum cost flow problem where each arc e in the original network has a cost of $\hat{\gamma}_e$ and each u-t path costs -$\hat{\alpha}_u$.

More formally, construct an auxiliary network $N^a = (V^a, A^a)$ as follows. For each node $v \in V$ we create a corresponding node $v^a \in V^a$. Similarly, for each arc $(i, j) \in A$ we create a corresponding arc $(i^a, j^a) \in A^a$ with a capacity of one unit and a cost per unit flow of $\hat{\gamma}_{(i,j)}$. In addition, we create another arc (u^a, s^a) with a lower bound of R + 1 units, an infinite capacity and a per-unit of flow cost of -$\hat{\alpha}_u$. We note that there is a bijection between every set of paths $P_{s-u} \in \mathcal{P}^R_{s-u}$ to the basic feasible solutions of the minimum cost flow problem on N^a. Moreover, we note that the cost of the flow in N^a equals the left-hand side of the inequality expressed in Equation 4.

The unit capacity on each arc in $A^a \backslash \{(u^a, s^a)\}$ ensures that the corresponding s-u paths are indeed arc-disjoint. The lower bound on arc (u^a, s^a) ensures that the set of s-u paths obtained from solving the minimum cost flow problem is of cardinality strictly greater than R. If there is no such set of paths, then the minimum cost flow problem is infeasible.

We may assume without loss of generality that the optimal solution contains no zero-cost, non-zero flow circulations. It is well known that a circulation on a network with m arcs can be decomposed into a cycle flow along at most m directed cycles in $O(m^2)$ time. See section 3.5 in Ahuja et al. [3].

If the optimal objective value of the minimum cost flow problem on network N^a is strictly less than -$\hat{\alpha}_u R$, then there is a violated source-to-node inequality and a corresponding optimal basic feasible solution gives a set of arc-disjoint paths, which, along with u, provides a violated inequality. Similarly, if the optimal objective value of the minimum cost flow problem on network N^a is greater than or equal to -$\hat{\alpha}_u R$, then all inequalities from Equation 3 are satisfied.

Thus, if we repeat this procedure for each node $u \in V$, we obtain a polynomial-time separation routine. $\square$

We shall henceforth refer to the LP relaxation of Wood's ILP with both the node-to-sink valid inequalities and the source-to-node inequalities as the *strengthened LP*.

2.3.3 Strength of Valid Inequalities

In this subsection, we provide an example of an optimal fractional extreme point to the LP relaxation of Wood's ILP that violates a source-to-node valid inequalities for a specific class of instances. A similar example for the node-to-sink valid inequalities can be easily constructed.

Let $\kappa \geq 2$ be a parameter. we construct a network $N_\kappa = (V_\kappa, E_\kappa)$ where V_κ is composed of three sets of nodes: $(\{s,t\}, X_\kappa, Y_\kappa)$ and E_κ is composed of three sets of arcs: $(E^s_\kappa, E^b_\kappa, E^t_\kappa)$. $|X_\kappa| = \kappa$ and $|Y_\kappa| = \mu - \kappa$, where μ is a parameter and a positive integer such that $\mu >> \kappa$.

We now describe the structure and capacity of the arcs in N_κ. For each node $v \in X_\kappa$, there exists an arc $e = (s,v) \in E^s_\kappa$ that has a capacity of μ units. Similarly, for each node $v \in X_\kappa \cup Y_\kappa$, there exists an arc $e = (v,t) \in E^t_\kappa$ that has a capacity of one unit. Finally, for each pair of nodes $u \in X_\kappa$ and $v \in Y_\kappa$, there exists an arc $(u,v) \in E^b_\kappa$ that has a capacity of μ^2 units. Since this is an instance of CMFNIP, all arcs have an interdiction cost of one unit.

We define the instance of CMFNIP $I_{w,\kappa}$ as minimizing the maximum flow on the network N_κ with an interdiction budget of $\kappa - 1$ units. See Figure 2 for an example of the network for instance $I_{w,\kappa}$.

First, we construct a fractional extreme point solution to the LP relaxation of Wood's

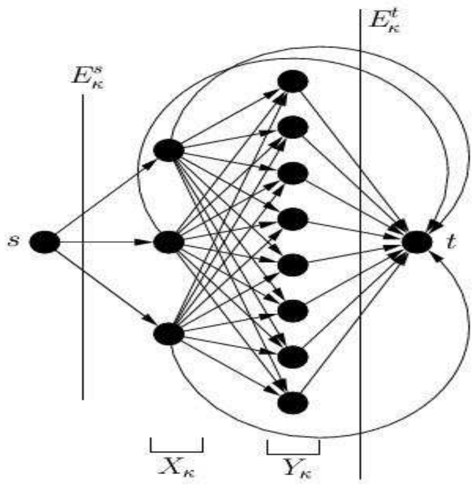

Figure 1: Network for CMFNIP instance $I_{w,\kappa}$ with $\kappa = 3$ and $\mu = 11$.

ILP.

Lemma 6. *There exists a feasible solution to the LP relaxation of Wood's ILP for CMFNIP instance $I_{w,\kappa}$ that has objective value of $\frac{\mu}{\kappa}$.*

Proof: It suffices to construct such a solution. Consider the following:

$\alpha_v = \frac{\kappa-1}{\kappa} \quad \forall\, v \in V \backslash \{s,t\}$

$\alpha_s = 0$

$\alpha_t = 1$

$$\beta_e := \begin{cases} \frac{\kappa-1}{\kappa} & \forall\, e \ \in E^s_\kappa \\ 0 & \text{otherwise.} \end{cases}$$

$$\gamma_e := \begin{cases} \frac{1}{\kappa} & \forall\, e \ \in E^t_\kappa \\ 0 & \text{otherwise.} \end{cases}$$

It can be verified that this solution is feasible and has objective value $\frac{\mu}{\kappa}$, since $c_e = 1 \ \forall\, e \in E^t_\kappa$. $\square$

Let $\hat{x}$ be the fractional solution constructed during the proof of Lemma 6.

To illustrate that $\hat{x}$ needs to be cut away when solving for the integer optimal solution, we state an integer optimal solution to instance $I_{w,\kappa}$ as a fact.

Fact 1. *An integer optimal solution to instance $I_{w,\kappa}$ may be constructed by arbitrarily interdicting $\kappa - 1$ arcs in the set E^t_κ. This solution has an objective value of $\mu - \kappa + 1$.*

For those interested in a proof of Fact 1, we refer the reader to the proof of Lemma 10 in the next section, which is very similar.

Lemma 7. *The fractional solution is cut away by a source-to-node inequality.*

Proof: Let $u \in Y_\kappa$ and let P_{s-u} be the set of maximum arc-disjoint s-u paths. Since $|P_{s-u}| = \kappa > R$, there exists a source-to-node valid inequality for the node u and the set of paths P_{s-u}.

Let $\hat{\alpha}_u$ denote the value of the decision value α_u in $\hat{x}$ and let $\hat{\gamma}_e$ denote the value of decision variable γ_e in $\hat{x}$ for each arc $e \in E$. Note that by construction of $\hat{x}$, $\hat{\gamma}_e = 0$ for each $e \in A(P_{s-u})$. Moreover, note that the term $(R - |P_{s-u}|)\hat{\alpha}_u$ equals -$\frac{\kappa-1}{\kappa}$, which is always strictly less than zero since $\kappa \geq 2$. Thus, as stated in Equation 3, the source-to-node valid inequality corresponding to node u and arc-disjoint path set P_{s-u} is violated. □

2.3.4 Using the Valid Inequalities in Practice

We tested these valid inequalities using the CPLEX 9.0 MIP solver. First of all, we note that most instances of CMFNIP, even those with hundreds of nodes, solve very quickly in practice without any modification to the default CPLEX settings. Thus, when testing both the source-to-node inequalities and the node-to-sink inequalities, we focused on instances of CMFNIP that had poor integrality gaps when using Wood's ILP.

We tested these two valid inequalities on instances $I_{w,\kappa}$ for a range of appropriately chosen values for the κ and $|V|$. Specifically, we added all violated node-to-sink inequalities and all violated source-to-node inequalities to the root node of the branch-and-bound tree. We tried adding the violated inequalities using the subroutine `CPXaddrows()` as well as using the subroutine `CPXaddusercuts()`. We did not subsequently add any other of these valid inequalities in any other node of the branch-and-bound tree besides the root node.

Table 1 contains a collection of prototypical results from our experimentation with these valid inequalities. For this class of deterministically constructed instances, there are definitely several cases where adding the cuts in some form reduces the running time needed for CPLEX to terminate with an optimal solution. However, there are several instances where adding the violated valid inequalities increased the running time of the MIP solver. Nevertheless, this experimentation suggests that these valid inequalities can be advantageous and should be considered as a possible technique to improve the running time for commercial integer programming solvers for difficult instances of CMFNIP.

2.4 Integrality Gap Result

In this section, we prove that the integrality gap of Wood's ILP, even with the strengthened LP relaxation, is contained in the set $\Omega(|V|^{1-\epsilon})$ where ϵ is any constant in $(0, 1)$ and V is the set of nodes in the CMFNIP network.

Claim 1. *Choose $\epsilon \in (0, 1)$. Then there exists an instance I of CMFNIP on a network $N = (V, A)$ with interdiction budget R where*

Table 1: Prototypical Results for MFNIP Valid Inequalities

NumNodes	Kappa	CutScheme	TotalTime	SeparationTime
200	30	none	27	0
200	30	CPXaddrows	27	2
200	30	CPXaddusercuts	101	3
200	50	none	107	0
200	50	CPXaddrows	192	5
200	50	CPXaddusercuts	87	3
200	75	none	82	0
200	75	CPXaddrows	334	5
200	75	CPXaddusercuts	167	6
250	25	none	36	0
250	25	CPXaddrows	87	4
250	25	CPXaddusercuts	239	4
250	62	none	240	0
250	62	CPXaddrows	643	9
250	62	CPXaddusercuts	192	10
300	30	none	162	0
300	30	CPXaddrows	747	7
300	30	CPXaddusercuts	465	6
300	75	none	905	0
300	75	CPXaddrows	607	20
300	75	CPXaddusercuts	> 57000	20
450	112	none	2435	0
450	112	CPXaddrows	2173	88
450	112	CPXaddusercuts	1682	86

$$\frac{z^*_{ILP}(I)}{z^*_{SLP}(I)} \in \Omega(|V|^{1-\epsilon}).$$

Here, $z^*_{ILP}(I)$ and $z^*_{SLP}(I)$ denote the optimal objective value of the ILP and the strengthened LP relaxation for instance I respectively.

To prove this claim, first we show that given a positive integer constant $\kappa \geq 2$, there exists an instance of CMFNIP I_κ such that $\frac{z^*_{ILP}(I_\kappa)}{z^*_{SLP}(I_\kappa)} \geq \kappa$. Then we discuss how to choose an appropriate κ in terms of ϵ to obtain the desired result.

Given $\kappa \geq 2$, we construct a network $N_\kappa = (V_\kappa, E_\kappa)$. The node set V_κ is partitioned

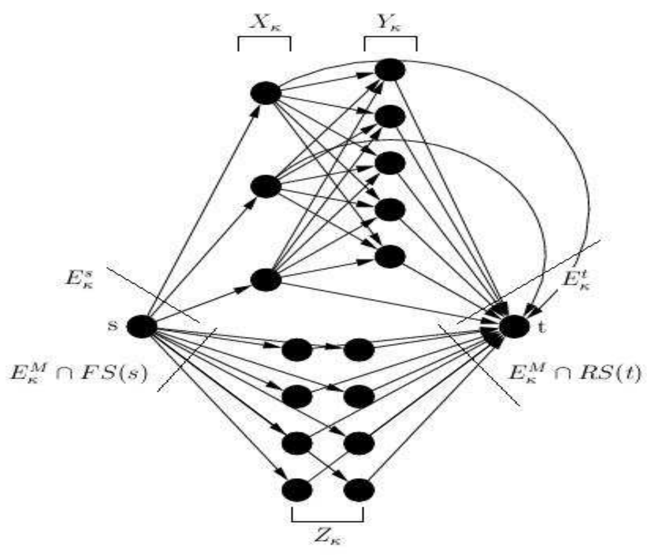

Figure 2: Network for CMFNIP instance I_κ with $\kappa = 3$ and $\mu = 8$.

into four sets of nodes X_κ, Y_κ, Z_κ and $\{s,t\}$ where $X_\kappa \cup Y_\kappa \cup Z_\kappa \cup \{s,t\} = V_\kappa$ and $S_i \cap S_j = \emptyset \; \forall \; S_i, S_j \in \{X_\kappa, Y_\kappa, Z_\kappa, \{s,t\}\}$ such that $S_i \neq S_j$. Given the parameter κ, the sizes of the first three respective node partitions are $|X_\kappa| = \kappa$, $|Y_\kappa| = \mu - \kappa$ and $|Z_\kappa| = \mu$, where μ is another parameter and a positive integer such that $\mu >> \kappa$. The arc set E_κ is partitioned into four sets of arcs E^s_κ, E^b_κ, E^t_κ and E^M_κ where $E^s_\kappa \cup E^b_\kappa \cup E^t_\kappa \cup E^M_\kappa = E_\kappa$ and $E^i_\kappa \cap E^j_\kappa = \emptyset \; \forall \; i,j \in \{s,b,t,M\}$ such that $i \neq j$.

We now describe the structure and capacity of the arcs in N_κ. For each node $v \in X_\kappa$, there exists an arc $e = (s,v) \in E^s_\kappa$ that has a capacity of μ units. Similarly, for each node $v \in X_\kappa \cup Y_\kappa$, there exists an arc $e = (v,t) \in E^t_\kappa$ that has a capacity of one unit. For each pair of nodes $u \in X_\kappa$ and $v \in Y_\kappa$, there exists an arc $(u,v) \in E^b_\kappa$ that has a capacity of μ^2 units. Finally, for each node $v \in Z_\kappa$ there exists both arcs $(s,v),(v,s) \in E^M_\kappa$ where each arc has a capacity of μ^2 units. Since this is an instance of CMFNIP, all arcs have an interdiction cost of one unit.

We define the instance of CMFNIP I_κ as minimizing the maximum flow on the network N_κ with an interdiction budget of $\mu + \kappa - 1$ units. See Figure 2 for an example of the network for instance I_κ.

We now prove a few lemmata about the optimal integer and fractional solutions to CMFNIP when using the ILP with the strengthened LP relaxation.

Lemma 8. *When using Wood's ILP for CMFNIP, there exists an optimal integer solution to the instance I_κ where $\beta_e = \gamma_e = 0 \; \forall \; e \in E^b_\kappa$.*

Proof: Suppose this is not true. Consider any optimal solution x_0^* that serves as a counterexample to Lemma 8 and let $E^{b^*}_\kappa \subseteq E^b_\kappa$ be the set of arcs where $\alpha_v = 1$ and either $\beta_{(u,v)} = 1$ or $\gamma_{(u,v)} = 1$ for all arcs $(u, v) \in E^{b^*}_\kappa$. Let $Y^*_\kappa \subseteq Y_\kappa$ be the set of nodes that are entered by at least one arc in $E^{b^*}_\kappa$.

We partition the nodes in Y^*_κ as follows: let $Y^{\beta^*}_\kappa = \{v \in Y^*_\kappa | \; \exists \; e \in RS(v) : \beta_e = 1\}$ and let $Y^{\gamma^*}_\kappa = Y^*_\kappa \backslash Y^{\beta^*}_\kappa$. We define a new solution x_1^* by keeping all of the variable values expressed in x_0^* except for setting $\beta_e = \gamma_e = 0$ for all arcs $e \in E^{b^*}_\kappa$, setting $\alpha_v = 0$ for all nodes $v \in Y^*_\kappa$, setting $\beta_{(v,t)} = 1$ for all nodes $v \in Y^{\beta^*}_\kappa$ and setting $\gamma_{(v,t)} = 1$ for all nodes $v \in Y^{\gamma^*}_\kappa$. It should be clear that both the objective value as well as the interdiction cost of x_1^* are less than or equal to those of x_0^*. Thus, x_1^* is both feasible and optimal. □

Lemma 9. *When using Wood's ILP for CMFNIP for instance I_κ, then for any optimal solution, $\gamma_e = 0$ for all arcs $e \in E^M_\kappa$*

Proof: First we construct a feasible solution. Then, given the existence of this feasible solution, we may conclude that $\gamma_e = 0$ for all arcs $e \in E^M_\kappa$, otherwise the optimal solution has an objective value greater than the feasible solution we construct.

Choose $E^{t\beta}_\kappa \subset E^t_\kappa$ such that $|E^{t\beta}_\kappa| = \kappa - 1$. Now consider the following solution:

$\alpha_v = 0 \quad \forall \; v \in V_\kappa \backslash \{t\}$

$\alpha_t = 1$

$$\beta_e := \begin{cases} 1 & \forall\ e\ \in E_\kappa^{t\beta} \cup E_\kappa^M \cap RS(t) \\ 0 & \text{otherwise.} \end{cases}$$

$$\gamma_e := \begin{cases} 1 & \forall\ e\ \in E_\kappa^{t} \backslash E_\kappa^{t\beta} \\ 0 & \text{otherwise.} \end{cases}$$

Since our interdiction budget $R = \mu + \kappa - 1$ and $|E_\kappa^M \cap RS(t)| = \mu$, this solution is clearly feasible. Furthermore, note that the objective value of the solution above is $\mu - \kappa + 1$.

Since the capacity of each arc in E_κ^M is μ^2 we see that any feasible solution with $\gamma_e = 1$ for some arc $e \in E_\kappa^M$ must have an objective value of at least μ^2 and therefore cannot be optimal. □

Lemma 10. *When using Wood's ILP for CMFNIP, the optimal objective value for instance I_κ is $\mu - \kappa + 1$.*

Proof: Lemma 8 demonstrates that there exists an optimal solution of Wood's ILP for CMFNIP that where $\beta_e = \gamma_e = 0$ for all arcs $e \in E_\kappa^b$. Thus, we may assume there exists an optimal solution that also has $\alpha_u = \alpha_v$ for all arcs $(u, v) \in E_\kappa^b$. Moreover, Lemma 9 indicates that each of the μ arc-disjoint paths formed by the arcs in E_κ^M must be interdicted, which requires μ units of interdiction resources. Thus, constructing an optimal solution reduces to deciding how to allocate the remaining $\kappa - 1$ units of interdiction resources between the arcs in $E_\kappa^s \cup E_\kappa^t$.

Since $\alpha_u = \alpha_v$ for all arcs $(u, v) \in E_\kappa^b$, note that there exists an optimal solution where either have $\beta_e = \gamma_e = 0$ for all $e \in E_\kappa^s$ or $\beta_e = \gamma_e = 0$ for all $e \in E_\kappa^t$. Since exactly one of these may be true, the s-t cut defined by the α variables is either equivalent to E_κ^s or equivalent to E_κ^t.

Since $\alpha_u = \alpha_v$ for all arcs $(u, v) \in E_\kappa^b$, we may assume that $\alpha_u = \alpha_v$ for all nodes u,

$v \in V_\kappa \backslash \{s,t\}$ without loss of generality. If the s-t cut defined by the α variables is $FS(s)$, then interdicting all arcs in $FS(s) \cap E_\kappa^M$ and removing any arbitrary subset of $\kappa - 1$ arcs from E_κ^s leaves a s-t cut of capacity μ, as there is exactly one arc in E_κ^s which could not be removed. Note that this is equal to the total capacity of all of the arcs in E_κ^t, which is μ.

If the s-t cut defined by the α variables is $RS(t)$, then interdicting all arcs in $RS(t) \cap E_\kappa^s$ and removing any arbitrary subset of $\kappa - 1$ arcs from E_κ^t leaves a s-t cut of capacity $\mu - \kappa + 1$, which is also the new minimum capacity s-t cut in the remaining network. Since all other possible integer solutions have an objective value greater than or equal to $\mu - \kappa + 1$, we may conclude that interdicting any arbitrary subset of $\kappa - 1$ arcs in the set E_κ^t and interdicting all arcs in $RS(t) \cap E_\kappa^M$ describes an optimal solution for CMFNIP instance I_κ. $\square$

Lemma 11. *There exists a feasible solution to the strengthened LP relaxation for CMFNIP instance I_κ that has an objective value of $\frac{\mu}{\kappa}$.*

Proof: It suffices to construct such a solution. Consider the following:

$$\alpha_v := \begin{cases} \frac{\kappa-1}{\kappa} & \forall\ v \in X_\kappa \cup Y_\kappa \\ 0 & \forall\ v \in \{s\} \cup Z_\kappa \\ 1 & v = t \end{cases}$$

$$\beta_e := \begin{cases} \frac{\kappa-1}{\kappa} & \forall\ e\ \in E_\kappa^s \\ 1 & \forall\ e\ \in E_\kappa^M \cap RS(t) \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_e := \begin{cases} \frac{1}{\kappa} & \forall\ e\ \in E_\kappa^t \\ 0 & \text{otherwise} \end{cases}$$

It can be verified that this solution is feasible and has objective value $\frac{\mu}{\kappa}$, since $c_e =$

$1 \; \forall \; e \in E^t_\kappa$. $\Box$

In fact, the constructed solution is an optimal solution for the strengthened LP relaxation. This can be verified from the complementary slackness conditions after constructing an appropriate dual feasible solution. However, demonstrating that this is a feasible solution suffices for our stated purpose.

Theorem 12. *Let $\mathcal{I}$ be the family of all instances of CMFNIP. For any $\epsilon \in (0, 1)$:*

$$sup_{I\in\mathcal{I}} \frac{z^*_{ILP}(I)}{z^*_{SLP}(I)} \in \Omega(|V_\kappa|^{1-\epsilon})$$

Proof: Consider the parameter μ and instance I_κ. From Lemma 11 we know $z^*_{SLP}(I_\kappa) \leq \frac{\mu}{\kappa}$ and from Lemma 10 we know $z^*_{ILP}(I_\kappa) = \mu - \kappa + 1$. Thus,

$$\frac{z^*_{ILP}(I_\kappa)}{z^*_{SLP}(I_\kappa)} \geq \frac{\mu - \kappa + 1}{\frac{\mu}{\kappa}}.$$

What remains to show are sufficient choices for κ, μ and $|V_\kappa|$. Recall that $\epsilon \in (0, 1)$. Let $\kappa = \lfloor |V_\kappa|^{1-\epsilon} \rfloor$ and let $|V_\kappa|$ be sufficiently large such that $\kappa \geq 2$. Note that by construction of I_κ, we have $\mu = \frac{|V_\kappa|}{2} - 1$. Substituting for μ and κ we get the following inequality:

$$\frac{z^*_{ILP}(I_\kappa)}{z^*_{SLP}(I_\kappa)} \geq \frac{\lfloor |V_\kappa|^{1-\epsilon} \rfloor (\frac{|V_\kappa|}{2} - 1 - \lfloor |V_\kappa|^{1-\epsilon} \rfloor + 1)}{\frac{|V_\kappa|}{2} - 1}.$$

Using the definition of Ω, we obtain:

$$\frac{z^*_{ILP}(I_\kappa)}{z^*_{SLP}(I_\kappa)} \in \Omega(\frac{\lfloor |V_\kappa|^{1-\epsilon} \rfloor (\frac{|V_\kappa|}{2} - 1 - \lfloor |V_\kappa|^{1-\epsilon} \rfloor + 1)}{\frac{|V_\kappa|}{2} - 1}) = \Omega(\frac{\lfloor |V_\kappa|^{1-\epsilon} \rfloor (\frac{|V_\kappa|}{2} - 1)}{\frac{|V_\kappa|}{2} - 1}).$$

Continuing with algebra, we obtain:

$$\frac{z^*_{ILP}(I_\kappa)}{z^*_{SLP}(I_\kappa)} \in \Omega(\frac{\lfloor |V_\kappa|^{1-\epsilon} \rfloor (\frac{|V_\kappa|}{2} - 1)}{\frac{|V_\kappa|}{2} - 1}) = \Omega(\lfloor |V_\kappa|^{1-\epsilon} \rfloor) = \Omega(|V_\kappa|^{1-\epsilon}).$$

Thus, we may conclude that $\frac{z^*_{ILP}(I_\kappa)}{z^*_{SLP}(I_\kappa)} \in \Omega(|V_\kappa|^{1-\epsilon})$. Since $I_\kappa \in \mathcal{I}$, we may conclude that $sup_{I \in \mathcal{I}} \frac{z^*_{ILP}(I)}{z^*_{SLP}(I)} \in \Omega(|V_\kappa|^{1-\epsilon})$. □

Corollary 13. *The integrality gap of Wood's ILP (without the strengthened LP relaxation) for CMFNIP (and therefore MFNIP as well) is in the set* $\Omega(|V_\kappa|^{1-\epsilon})$ *for any* $\epsilon \in (0, 1)$.

Proof: The proof follows immediately from the fact that the plain LP relaxation of Wood's ILP for CMFNIP is a relaxation of the strengthened LP relaxation. Furthermore, since CMFNIP is a special case of MFNIP, this result extends to MFNIP as well. □

2.5 Hardness of Approximation of MFNIP

An immediate corollary of Theorem 12 is the following:

Corollary 14. *It is NP-hard to obtain an* $O(|V_\kappa|^{1-\epsilon})$*-factor approximation algorithm for CMFNIP (and MFNIP) that uses the strengthened LP relaxation as a lower bound.*

Corollary 14 leaves open the question, is it NP-hard to obtain an $O(|V_\kappa|^{1-\epsilon})$-approximate solution to CMFNIP for any arbitrary lower bound? Although the authors are unable to answer this question at the time this has been written, we have offered insight towards resolving this question for the general case of MFNIP, which we detail in this section.

Specifically, we present an *approximation-factor-preserving reduction* from RIC to MFNIP, meaning that any hardness of approximation result on RIC will immediately extend to MFNIP. First, we define the R-Interdiction Covering Problem (RIC). Then we prove that RIC is strongly NP-hard. Finally, we give the approximation-factor-preserving reduction.

R-Interdiction Covering Problem: Given a bipartite graph $G = (V_f, V_s, E)$ where the vertices in the partition V_f correspond to *facilities*, the vertices in the partition V_s correspond to *satellites* and E is the set of edges in G. A facility $u \in V_f$ serves a satellite $v \in V_s$ if and only if $(u, v) \in E$. Given a budget of R units, which R facilities should an *interdictor* remove from G so as to maximize the number of satellites who are not adjacent to any facilities in the resulting graph?

In the problem statement above, we use the term *graph* to denote a network where arcs lack orientation. We call an arc with no orientation an *edge*.

RIC was first stated in [21] and has applications in identifying critical infrastructure in supply (e.g., food, energy, medicine), domestic service (e.g., police, fire, EMS) or communication networks [21].

Lemma 15. *RIC is strongly NP-hard.*

Proof: We present a reduction from the Maximum Clique Problem, which is strongly NP-hard [34]. We note that our proof is a modification of the proof of Theorem 2 from [77].

Consider an arbitrary instance of the Maximum Clique Problem on a graph $G^q = (V^q, E^q)$. We show that the decision version of this problem: "Does the graph G^q contain a clique of size K?" is answered in the affirmative if and only if the decision version of RIC on a corresponding graph $G^R = (V_f^R, V_s^R, E^R)$: "Can we remove $R = |E^q| - \binom{K}{2}$ facilities from V_f^R to disconnect $|V_s^R| - K$ satellites in the graph G^R?" is answered in the affirmative.

First, we discuss how to construct the graph G^R from G^q. For each vertex $v \in V^q$, we create a satellite $s_v \in V_s^R$. Similarly, for each edge $(u, v) \in E^q$ we create a facility $f_{(u,v)} \in V_f^R$ and we add edges $(f_{(u,v)}, s_u)$ and $(f_{(u,v)}, s_v)$ to E^R.

Suppose that there is a clique of size K in G^q, say denoted by subgraph $G_c^q = (V_c^q, E_c^q)$. Then, by removing R facilities from V_f^R, we can disconnect $|V_s^R| - K$ satellites by interdicting $f_e \; \forall \; e \in E^q \backslash E_c^q$.

Similarly, suppose that we can remove R facilities from G^R to disconnect all but K satellites. Then the $\binom{K}{2}$ facilities that were not interdicted correspond to $\binom{K}{2}$ unique edges in E^q that are between the K vertices in V^q that uniquely correspond to the K satellites that were not disconnected in G^R. Since we have $\binom{K}{2}$ distinct edges between K vertices in a simple graph, then this must form a clique of size K, which is contained in G^q. The reduction is complete. $\square$

Theorem 16. *There is an approximation-factor-preserving reduction from RIC to MFNIP.*

Proof: Given an arbitrary instance of RIC I_f, we construct a corresponding instance of MFNIP I_n. Let $G = (V_f, V_c, E)$ be the graph in our instance of RIC. We construct an instance of MFNIP on the network $N = (V, A)$ as follows: for every vertex $v \in V_f \cup V_c$, there is a corresponding node $\bar{v} \in V$. Similarly, for every edge $e = (v_f, v_c) \in E$ there is a corresponding arc $\bar{e} = (\bar{v}_f, \bar{v}_c) \in A$ that originates from $\bar{v}_f$, terminates in $\bar{v}_c$, has unit capacity and has interdiction cost $R + 1$.

Let $\bar{V}_f$ be the set of nodes in V corresponding to facilities in V_f and let $\bar{V}_c$ be defined similarly. V also contains a source s and a sink t, either of which does not correspond with any vertices in $V_f \cup V_c$. For each node $\bar{v}_f \in \bar{V}_f$, there is an arc $(s, \bar{v}_f) \in A$ with capacity equal $|\{c \in V_c : (f, c) \in E\}|$ and an interdiction cost of one unit. Similarly, for each node $\bar{v}_c \in \bar{V}_c$, there is an arc $(\bar{v}_c, t) \in A$ with a capacity of one unit and an interdiction cost of $R + 1$. The interdiction budget for I_n equals the interdiction

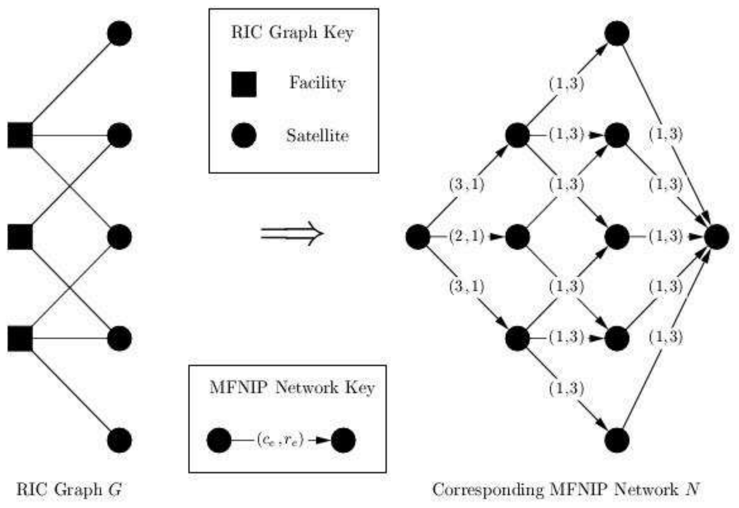

Figure 3: Approximation-factor-preserving reduction from an instance of RIC with $|V_f| = 3$, $|V_c| = 5$ and $R = 2$ to MFNIP.

budget in I_f, which is R.

It should be clear that RIC instance I_f has a solution with an objective value of z if and only if MFNIP instance I_n has a uniquely corresponding solution with an objective value of z. Thus, if it is NP-hard to obtain an α-factor approximate optimal solution for RIC, then it is NP-hard to obtain an α-factor approximate optimal solution for MFNIP. The reduction is complete. $\square$

See Figure 3 for an illustration of an example of the approximation-factor-preserving reduction described above.

2.6 Conclusions and Future Work

In this chapter, we have made several contributions on both CMFNIP and MFNIP. First, we have discovered two new exponentially large classes of valid inequalities for CMFNIP and provided polynomial-time separation algorithms for each.

Second, we have identified and proved that the integrality gap of Wood's ILP for CMFNIP (and therefore MFNIP) is contained in the set $\Omega(|V|^{1-\epsilon})$, even when the LP relaxation of Wood's ILP is strengthened with our valid inequalities. We are confident that this offers general problem insight. Specifically, when other researchers sought to compute the optimal objective value for MFNIP for each possible value for the interdiction budget (e.g., [11], [67] and [74]), the authors indicated that a small number of possible values for the interdiction budget were "problematic". Conceptually, a "problematic" budget is one where the corresponding instance of MFNIP takes an unusually long amount of time to obtain an optimal solution using their integer programming approach. Given the potential for a rather large integrality gap, which is a function of the interdiction budget, our result offers an explanation as to why "problematic" values for the interdiction budget may arise.

In addition, our result implies that it is NP-hard to find a $O(|V|^{1-\epsilon})$-approximate optimal solution of MFNIP when using the strengthened LP relaxation as a lower bound. We leave the question of if this is true for any arbitrary lower bound as an open question. We also pose the related open question:

Open Question: Does there exist a polynomial-time, constant factor approximation algorithm for the Maximum Flow Network Interdiction Problem?

Although we are unable to solve this question, we provide insight towards this endeavor. Specifically, we prove that any hardness of approximation result on RIC, an interdiction problem that is much simpler in problem structure, immediately extends to MFNIP.

CHAPTER III

LOCAL SEARCH IN MAXIMUM FLOW NETWORK INTERDICTION

In this chapter, we discuss our work on local search in maximum flow network interdiction. In the first section, we motivate this work and overview relevant literature. In the second section, we demonstrate the sensitivity of the computation time on the interdiction budget. In the third section we detail our neighborhood and our corresponding meta-heuristic search. In the fourth section we present our computational results for our meta-heuristic. In the final section we draw conclusions and discuss extensions of this work.

3.1 Introduction

Most of the previous computational work on MFNIP uses integer linear programming (ILP) techniques. In [77], Wood introduces a new ILP formulation, proves that it is equivalent to the intuitive "min-max" formulation and provides a new class of valid inequalities. In [25], Derbes heuristically solves MFNIP using Lagrangian relaxation combined with a binary search on the Lagrange multipliers. Both [11] and [74] expand on Derbes by solving MFNIP for all possible interdiction budgets; thus implicitly constructing the Pareto-efficient frontier. Finally, in [67], Royset and Wood construct an approximate Pareto-efficient frontier using an implicit enumeration of all near minimum capacity cuts.

In [24], Dai and Poh conducted a small study of using a simple genetic algorithm to solve MFNIP. However, the documented study is very limited. Their experiments are briefly summarized at the beginning of our computational results section.

All of the previous work on MFNIP, with the exception of [24], uses Wood's formulation. All integer programming results in this chapter are with respect to Wood's formulation.

In both [11] and [74] the authors indicate that certain values of the interdiction budget proved to be "problematic" for an otherwise fixed instance of MFNIP, meaning that the employed Lagrangian approaches failed to obtain an optimal solution for those budget values. However, there was no further discussion on why these "problematic" budgets may exist, how often they occur or how one may recognize them in advance.

In this chapter, we demonstrate that the time required to solve a MFNIP instance is highly sensitive on the interdiction budget. This is an important observation for several reasons. First, even if standard commercial software can obtain optimal solutions for most instances in a reasonable amount of time, it does not address the issue of when a very important instance might be in the minority of instances that cannot be completely solved in a reasonable amount of time. Second, in the context of the motivating applications of MFNIP such as intercepting smuggled drugs or nuclear material, policymakers have the ability to change the available interdiction budget. Thus, there is value in knowing how varying the interdiction budget will impact the solution time of one's algorithm.

In addition to being the first researchers to explicitly document this, we undergo a theoretical study to explain this behavior. To this end, we observe how both the number of optimal solutions fluctuates with respect to the interdiction budget for both Wood's ILP and a natural LP relaxation. Furthermore, Corollary 13 proves that the integrality gap of Wood's ILP is not bounded below by a constant factor even in the case when all arcs have unit interdiction cost. This is the first known result on this widely used formulation. If the integrality gap is very large, then this suggests that lower bounds obtained from linear programming relaxations may not always be very useful.

In this chapter, we present a neighborhood for MFNIP based on evaluating cut-sets as well as a meta-heuristic framework for computing good heuristic solutions. Designing the neighborhood was nontrivial because computing the objective value of a feasible solution, which is a set of arcs to interdict, requires the solution to a maximum flow problem. Thus, there is no intuitive neighborhood for MFNIP that is also rapidly searchable in practice.

We demonstrate that our meta-heuristic is robust in three respects. First, the time it requires to obtain a good solution is relatively insensitive to the interdiction budget, unlike the time that ILP software requires to obtain a provable optimal solution. Second, the time required by our meta-heuristic is comparatively unaffected by the density of the network in the instance of MFNIP. Third, our meta-heuristic can obtain good quality solutions even when a very poor initial solution happens to be randomly selected. This is the first extensively documented local search approach to MFNIP.

Our neighborhood is also very advantageous in its modularity. It can be extended to more complicated maximum flow interdiction problems, such as the multiple resource interdiction problem described in [77], as long as a black-box solver is available to evaluate the subproblems that arise. In the case of multiple resource interdiction, a rapid heuristic for a multidimensional knapsack problem would be needed. Efficient computational approaches to this problem are detailed in [64].

To summarize, this chapter offers several contributions. First, we offer empirical evidence to explain the aforementioned time sensitivity. Second, we present the first published result on the integrality gap of Wood's formulation. Third, we present the first extensive neighborhood search approach to MFNIP. Lastly, we demonstrate a robust meta-heuristic search that is insensitive to the interdiction budget.

3.2 Time Sensitivity on the Interdiction Budget

Through computational testing, we observe that the time ILP software requires to solve instances of MFNIP, when using Wood's formulation, is highly sensitive to the interdiction budget. More formally, given a fixed topology with fixed capacities and fixed interdiction costs on the arcs, the time such software needs to solve an instance of MFNIP is highly sensitive on the chosen interdiction budget.

Figure 4 illustrates the computational sensitivity of an instance on its budget. The abscissa corresponds to the interdiction budget. The ordinate axis corresponds to the solution time. Note the logarithmic scale on the ordinate axis. The plotted line is the number of seconds it takes CPLEX to solve this particular 200 node instance of MFNIP. The instance below was randomly generated where each arc, on the complete directed graph with 200 nodes had a 50% probability of appearing. Given that an arc appeared, its flow capacity and interdiction costs were each chosen, uniformly at random, from the interval $[10, 20]$ with a correlation coefficient of .9.

We introduce a few definitions to clarify the subsequent content.

Definition 1. *Given a feasible solution to MFNIP, we say an arc* (u, v) *is* exposed *if* $\alpha_u - \alpha_v = 1$. *Similarly, given a feasible solution to the linear programming relaxation of Wood's formulation for MFNIP, we say an arc* fractionally exposed *if* $\alpha_u - \alpha_v > 0$.

Analogously, *exposed cuts* and *fractionally exposed cuts* are *s*-*t* cuts whose arcs are all either exposed or fractionally exposed respectively. If any (fractionally) exposed cut appears in an optimal solution, we describe it as an *optimal (fractionally) exposed cut.* A fractionally exposed arc that does not contribute to the objective value is said to be *fractionally interdicted.*

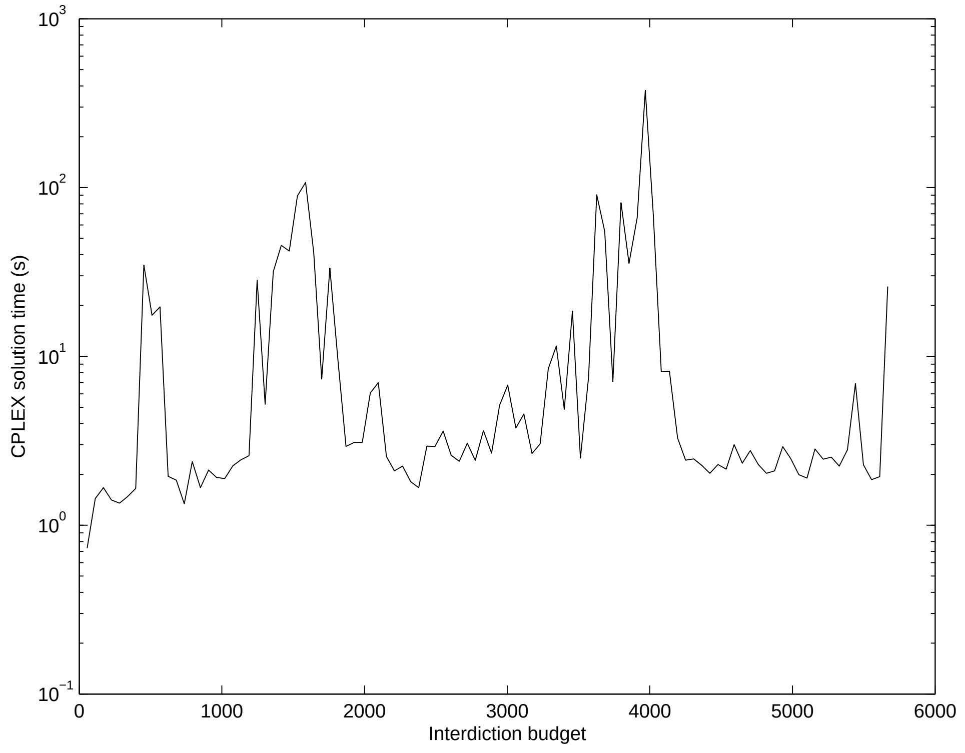

Figure 4: The computational sensitivity of an instance on its interdiction budget.

The number of optimally exposed s-t cuts (and likewise the number of optimal fractionally exposed s-t cuts for the LP relaxation) greatly depends on the interdiction budget; it is even possible for there to be exponentially many such cuts.

Figure 5 illustrates how the number of optimally exposed cuts depends on the interdiction budget. Suppose that all arcs have unit capacity. When the interdiction budget is 2, any s-t cut is an optimally exposed cut. When the interdiction budget is 1, there are only half as many. Namely, only the cuts that involve the arc with unit interdiction cost. We note that in either case, there are $O(2^n)$ optimally exposed cuts in this example.

A problem instance with a large number of optimal fractionally exposed cuts may require the traversal of a larger branch-and-cut tree as there would be more fractional extreme point solutions to cut away. Since the number of such cuts greatly varies

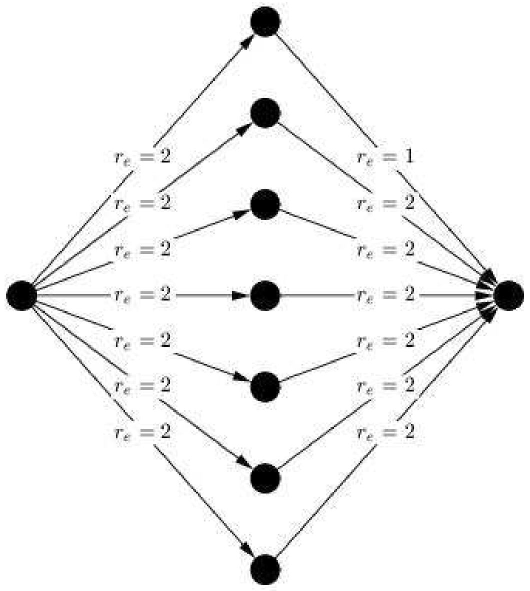

Figure 5: The number of optimally exposed cuts depends on the interdiction budget.

with the budget, this offers an explanation for the behavior illustrated in Figure 4. In the next section, we discuss the integrality gap of Wood's formulation and its relation to the time sensitivity on the interdiction budget.

3.3 Node-Flip Neighborhood

One of the main goals of this chapter is to design a neighborhood that is computationally insensitive to the interdiction budget. The neighborhood used is inspired by viewing MFNIP from the perspective of minimizing the minimum cut and developed from the following proposition:

Proposition 17. *Given an arbitrary instance of MFNIP, there exists an optimal solution where all of the interdicted arcs are contained in a s-t cut.*

Proof: If all of the interdicted arcs cannot be contained in a cut, then there must exist a vertex v that has at least one interdicted arc in both its forward star and its reverse star. We note that the throughput of v in the aftermath network equals

$\min\{\sum_{u \in RS(v)} c_{uv}, \sum_{u \in FS(v)} c_{vu}\}$. Without loss of generality, if we assume that the capacity of the forward star is less, then any arc in the reverse star that is interdicted may be added to the aftermath network without increasing the maximum flow. □

3.3.1 Storing and Evaluating Solutions

Since an optimal solution always exists to a combinatorial optimization problem, Proposition 17 implies that there exists a cut C^* that contains the optimal set of arcs to interdict, I^*. We implicitly store feasible solutions by instead storing cuts. To this end, we prove the following proposition:

Proposition 18. *Assume we are given an instance of MFNIP under the restriction that only arcs in a cut C may be interdicted. Then the optimal solution may be determined by solving a knapsack problem.*

Proof: Given any arbitrary cut C, if we only allow arc interdictions within this cut, we can determine the best set of arcs to interdict by solving the following knapsack problem:

$$z_C^* = \max \left\{ \sum_{a \in C} c_a x_a : \sum_{a \in C} r_a x_a \leq R,\ x_a \in \{0,1\}. \right\} \tag{5}$$

Thus, the capacity of C in the aftermath network is:

$$\mathit{value}(C) = \sum_{a \in C} c_a - z_C^*. \tag{6}$$

□

Given a cut C, we let $K(\cdot) : \mathcal{C} \longrightarrow \mathcal{S}$ denote the function that maps a cut $C \in \mathcal{C}$ to a feasible solution $s \in \mathcal{S}$ by optimally solving the knapsack problem 5. Here, $\mathcal{C}$ denotes the set of all cuts in the original network and $\mathcal{S}$ denotes the set of all feasible solutions to our instance of MFNIP. This function will hereby be called the *knapsack function.*

It is well known that the 0-1 Knapsack Problem is weakly NP-hard [34]. We will discuss how we evaluate these knapsack problems efficiently later in the chapter.

Given a cut C, we describe the feasible solution $K(C)$ obtained from our knapsack function as *containable.* Given a containable feasible solution s, we describe the cut $K^{-1}(s)$ as its *cut representation.* Note that not all feasible solutions to an instance of MFNIP are necessarily containable and therefore lack a cut representation.

3.3.2 Neighborhood Function

In the interest of clarity, we will first discuss what an application of a flip neighborhood would be to MFNIP.

Recall that our feasible solutions are stored as cuts. Given an arbitrary feasible solution s_0 to an instance of MFNIP $\mathcal{I}$, let $C(s_0) = (S_0, T_0)$ be its representation as a cut. A *move* for the flip neighborhood would be to:

1. Transfer a non-terminal node from S_0 to T_0 or vice-versa. This creates a new cut C_1.
2. Obtain a new feasible solution to $\mathcal{I}$ from our knapsack function $K(C_1)$.

We refer to a move for the flip neighborhood as a *flip.* It is straightforward to construct the transition graph for this neighborhood. In the next subsection, we will refer to a neighborhood *transition network.* To convert a transition graph into a transition network, one merely needs to replace every edge (u, v) with two arcs: (u, v) and (v, u). We are now ready to define our neighborhood:

Definition 2. *Let s_0 be an arbitrary feasible solution to an instance $\mathcal{I}$ of MFNIP* *and let $\mathcal{S}$ denote the set of all feasible solutions to $\mathcal{I}$. Then solution s_1 is a* neighbor *of s_0 with respect to the* depth-k flip neighborhood *if and only if there exists a path from the vertex corresponding to solution s_0 to the vertex corresponding to solution s_1 of length at most k.*

3.3.3 Searching the Neighborhood

Given a parameter k, we employ an inexact neighborhood search technique on the depth-k neighborhood function. Specifically, for a current solution S_i, we will construct a sub-network of the transition network and choose the "best" solution in the sub-network to be our next solution S_{i+1}. For simplicity of description, we will often write "S_i" instead of "the node in the transition network corresponding to solution S_i".

While constructing the sub-network, we impose the following restrictions:

1. The sub-network must be a connected tree that is rooted at S_i.
2. The sub-network cannot contain more than a pre-specified number of nodes.
3. The length of the path between S_i and $\tilde{S}_j$ must be at most k for all solutions $\tilde{S}_j$ in the sub-network. This is called the *depth* requirement.
4. The number of solutions in the sub-network that are of depth ℓ cannot exceed a pre-specified amount. This is called the *breadth* requirement.

We refer to the sub-network constructed in this fashion as the *transition tree.*

It remains to be explained how we choose which solutions to insert into the transition tree. Throughout the construction of the transition tree, we maintain a binary heap of potential tree solutions. We refer to this heap as our potential solution *pool.* In addition, given a solution S_i we refer to any adjacent solution $\tilde{S}_j$ in the original

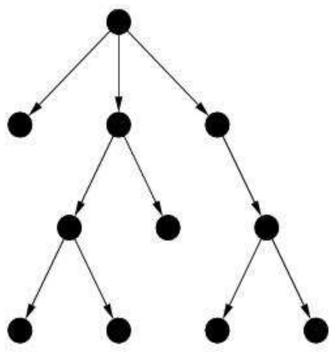

Figure 6: Solution tree that has a depth of 3 and a breadth of 4.

transition graph as a *flip-neighbor*. We note that this relationship is different from what a neighbor would be in the k-depth flip neighborhood.

After a solution is added to the transition tree, we add a subset of its flip-neighbors to the solution pool. In practice, we found it most efficient to select these neighbors at random. It is straightforward to see that any solution in the transition tree may not have more children than the maximum breadth requirement for the tree. If w is the maximum breadth allowed, then we specifically add w flip-neighbors to the pool.

When a flip-neighbor of S_i is inserted into the potential solution pool, the corresponding object contains several pieces of information. In addition to the cut representation of the solution, the object also contains a pointer to the parent node of the solution, if the solution were to be inserted into the transition tree, that is, a pointer to S_i. In addition, the object contains a *key* for the solution pool (which is a binary heap). The key is an estimate on objective value of the MFNIP solution. Using equation (6), we can compute this value by solving a knapsack problem. However, in practice, solving a knapsack problem for each potential solution is too time consuming. In the interest of celerity, we elected to use a rapid $\frac{1}{2}$-approximation algorithm for the knapsack problem found in [50].

Algorithm 1 provides the pseudocode for inserting flip-neighbors into the solution pool.

Algorithm 1 Put Flip Neighbors In Pool

```
putRandomNeighborsInPool(feasible solution $S$, potential tree solution pool $P$)
  $\#flips \leftarrow 0$
  while $\#flips < maxBreadthAllowed$ do
    randomly pick a node $i \in V \setminus \{s, t\}$
    $\#flips \leftarrow \#flips + 1$
    if $i \in source(S)$ then
      $source(S') \leftarrow source(S) \setminus \{i\}$
      $sink(S') \leftarrow sink(S) \cup \{i\}$
    else
      $source(S') \leftarrow source(S) \cup \{i\}$
      $sink(S') \leftarrow sink(S) \setminus \{i\}$
    end if
    $key(S') \leftarrow$ compute approximate value of $S'$
    insert $S'$ into pool of potential tree solutions $P$
  end while
```

At each iteration of our local search, we will construct a new transition tree starting from a solution S_i. We continue to construct a transition tree until either we have reached the maximum number of solutions allowed in the tree or it is not possible to add another solution to the tree without violating any of the aforementioned restrictions. Once we finish constructing the transition tree, we choose the best node as the root for a new transition tree to be constructed during the next iteration.

While the construction of the transition tree is not complete, we remove the solution from the pool with the lowest heap key, say $\tilde{S}_j$. We add $\tilde{S}_j$ to the transition tree if it is not already in the tree and adding the node would not violate our breadth restriction. If $\tilde{S}_j$ is not at the maximum allowable depth and its addition does not complete the construction of the tree, then we add a random subset of flip-neighbors of $\tilde{S}_j$ to our pool.

After we add a solution to the transition tree, we compute its objective value by evaluating the knapsack function using the code of Pisinger, detailed in [62].

To avoid adding duplicate solution to the tree, we store pointers to all of the solutions in the tree in an array of linked lists. The objective value of the solution serves as a hash function from a feasible MFNIP solution to a linked list. Given a solution is mapped to a non-empty linked list, the list is then traversed to explore for identical solutions. The overall reduction in computation time from this procedure in practice is enormous.

Constructing the transition tree is a parameter driven approach. Restricting the depth of the tree allows for many nodes to be flipped in a single iteration. Allowing a greater breadth allows for more moves that might initially appear to be undesirable but may lead to better solutions in the future. Increasing the maximum transition tree size reduces the number of times a solution is re-examined but at the expense of requiring more memory.

Algorithm 2 contains our pseudocode for constructing the transition tree, given a solution S^{root}.

Algorithm 2 Construct Transition Tree

```
constructTree(feasible solution $S^{root}$)
  initialize transition tree $T$ with $S^{root}$ as root node
  initialize pool of potential tree solutions $P$

  putRandomNeighborsInPool($S^{root}, P$)    // start pool with neighbors of the root
  node
  while $size(T) < maxSizeAllowed$ do
    $S \leftarrow$ solution with lowest key value in $P$
    if ($S$ is not in $T$) or ($breadth(S,T) \leq maxBreadthAllowed$) then
      compute exact value of $S$
      insert $S$ into $T$
      if $depth(S,T) < maxDepthAllowed$ then
        putRandomNeighborsInPool($S, P$)
      end if
    end if
  end while
  $S^{best} \leftarrow$ best solution in transition tree $T$
  return $S^{best}$
```

We iteratively perform this neighborhood search procedure until the best solution in a constructed transition tree is its root. Since this is an inexact neighborhood search, this solution is not necessarily locally optimal with respect to our k-depth flip neighborhood. We hereby refer to these points as *inexact locally optimal solutions.*

3.3.4 Meta-Heuristic Search

To discuss our meta-heuristic framework, we must first discuss how to escape from inexact locally optimal solutions. To do this, we will randomly flip a pre-specified number of non-terminal nodes. Specifically, let λ denote the percentage of non-terminal nodes that will be flipped and let n be the number of nodes. Then we will choose $\lfloor \lambda \cdot (n-2) \rfloor$ nodes uniformly at random to be flipped. We refer to this function as perturbSolution().

Our overall procedure is as follows. We start with an initial cut, which we use to construct an initial solution. We then perform our inexact search of the k-depth flip neighborhood, where k is an user-specified parameter. This terminates with an inexact locally optimal solution S'. If this is the best solution we have seen throughout the entire search procedure, we record this solution as S^{best}. If we have not met a pre-specified termination criteria, then S' is modified using perturbSolution().

We adjust the parameter λ over the course of our meta-heuristic search. Specifically, if there is no improvement to the best known solution after executing K calls to performLocalSearch() then λ is multiplied by another user-specified *discount factor* ρ, where $0 < \rho < 1$. When this occurs, we say that λ has been *discounted.* The discounting of our perturbation here is analogous to cooling in simulated annealing.

The meta-heuristic search terminates after we have performed a pre-specified number of iterations L, including those immediately after λ has been discounted, without encountering a solution better than S^{best}. Algorithm 3 contains pseudocode of our meta-heuristic search for MFNIP.

Algorithm 3 Meta-Heuristic Search for MFNIP

```
perform preprocessing
construct initial solution $S^{init}$

$S^{best} \leftarrow S^{init}$
$S \leftarrow S^{init}$
$discountCounter \leftarrow 0$
$terminationCounter \leftarrow 0$

**while** $terminationCounter < L$ **do**
    $S' \leftarrow$ performLocalSearch($S$)
    **if** $value(S') < value(S^{best})$ **then**
        $S^{best} \leftarrow S'$
        $discountCounter \leftarrow 0$
        $terminationCounter \leftarrow 0$
    **else**
        $discountCounter \leftarrow discountCounter + 1$
        $terminationCounter \leftarrow terminationCounter + 1$
    **end if**
    **if** $discountCounter \geq K$ **then**
        $discountCounter \leftarrow 0$
        $\lambda \leftarrow \rho \cdot \lambda$
    **end if**
    $S \leftarrow$ perturbSolution($S', \lambda$)
**end while**
**return** $S^{best}$
```

3.4 Computational Experiments

3.4.1 Previous Computational Settings

The only other study on local search in MFNIP can be found in [24]. Here, Dai and Poh have studied a simple genetic algorithm on two small instances of MFNIP. On a 20-node instance, the genetic algorithm found the optimal solution on four of the ten runs reported. On a 50-node instance, the genetic algorithm only found the optimal solution on two of the ten runs reported. For this instance, the optimal objective value was 5 while the 8 sub-optimal executions of the meta-heuristic terminated with a best objective value of at least 14.

3.4.2 Experimental Settings

To generate instances, we will often fix a network topology, arc interdiction costs, arc capacities and generate an instance for each nontrivial budget value over an appropriate range. Although we are solving instances over a large range of budgets, our goal is not to compute the Pareto-efficient frontier (PEF). Had we intended to compute the PEF, we would be exploiting information obtained from previously solved instances.

In order to evaluate the effectiveness of our meta-heuristic, we compare its performance with that of CPLEX, using both solution quality and solution time as performance metrics. We conducted our experiments on a large set of randomly generated instances with a variety of network topologies, capacities, and interdiction costs.

Recall that an instance of MFNIP consists of a network topology, arc capacities, arc interdiction costs, and an interdiction budget. Each instance in our test set has a network topology on 200 nodes, randomly generated in one of the five ways listed below:

1. `arc` - Each arc has a 50% probability of appearing.
2. `path1` - A random topology is generated by appending random s-t paths until each node has in-degree of at least 1 and out-degree of at least 1.
3. `path5` - Similar to `path1`, except each node has in-degree of at least 5 and out-degree of at least 5.
4. `tpath1` - Similar to `path1`, except all s-t paths follow a fixed topological order.
5. `tpath5` - Similar to `path5`, except all s-t paths follow a fixed topological order.

Given a network topology, we randomly generated arc capacities using uniform distributions from three possible sets: $\{10, \ldots, 20\}, \{50, \ldots, 100\}, \{500, \ldots, 1000\}$.

Finally, given a network topology, we randomly generated arc interdiction costs using twelve possible distributions. Some methods involved correlation with the arc capacities, and some involved correlation with the distance of the arc to the source and sink. The first six distributions we used are as follows:

1. Positive correlation with minimum distance from source or sink, as follows:

$$c_{ij} = \left[\underline{c} + \frac{d_{ij} - \underline{d}}{\overline{d} - \underline{d}} (\overline{c} - \underline{c}) + \epsilon \right] \quad \text{for all } (i, j) \in A$$

where $[\cdot]$ is the rounding function,

$$d_{ij} = \min \left\{ \begin{array}{cc} \text{length of shortest} & \text{length of shortest} \\ s\text{-}i \text{ path in } N & j\text{-}t \text{ path in } N \end{array} \right\} \quad \text{for all } (i, j) \in A,$$

(the length of any i-j path is simply the number of arcs along the path), and

$$\underline{d} = \min\{d_{ij} : (i, j) \in A\}, \qquad \overline{d} = \min\{d_{ij} : (i, j) \in A\}, \qquad \underline{c} = 11, \qquad \overline{c} = 19,$$

and ϵ is uniformly distributed on $\{-1, 0, 1\}$. Note that $c_{ij} \in \{10, \ldots, 20\}$ for all $(i, j) \in A$.

2. Same as item 1, except $\underline{c} = 55$, $\overline{c} = 95$, and ϵ is uniformly distributed on $\{-5, \ldots, -1, 0, 1, \ldots, 5\}$. Note that $c_{ij} \in \{50, \ldots, 100\}$ for all $(i, j) \in A$.

3. Same as item 1, except $\underline{c} = 550$, $\overline{c} = 950$, and ϵ is uniformly distributed on $\{-50, \ldots, -1, 0, 1, \ldots, 50\}$. Note that $c_{ij} \in \{500, \ldots, 1000\}$ for all $(i, j) \in A$.

4. Negative correlation with minimum distance from source or sink, as follows:

$$c_{ij} = \left[\overline{c} - \frac{d_{ij} - \underline{d}}{\overline{d} - \underline{d}}(\overline{c} - \underline{c}) + \epsilon\right] \quad \text{for all } (i, j) \in A$$

   where $[\cdot]$, d_{ij} for all $(i, j) \in A$, $\underline{d}$, and $\overline{d}$ are all as defined in item 1, $\underline{c} = 11$, $\overline{c} = 19$, and ϵ is uniformly distributed on $\{-1, 0, 1\}$. Note that $c_{ij} \in \{10, \ldots, 20\}$ for all $(i, j) \in A$.

5. Same as item 4, except $\underline{c} = 55$, $\overline{c} = 95$, and ϵ is uniformly distributed on $\{-5, \ldots, -1, 0, 1, \ldots, 5\}$. Note that $c_{ij} \in \{50, \ldots, 100\}$ for all $(i, j) \in A$.

6. Same as item 4, except $\underline{c} = 550$, $\overline{c} = 950$, and ϵ is uniformly distributed on $\{-50, \ldots, -1, 0, 1, \ldots, 50\}$. Note that $c_{ij} \in \{500, \ldots, 1000\}$ for all $(i, j) \in A$.

We also used three distributions where interdiction costs were uniformly drawn, completely independent from arc capacities, from the following three sets: $\{10, \ldots, 20\}$, $\{50, \ldots, 100\}$ and $\{500, \ldots, 1000\}$. Lastly, we used three distributions where interdiction costs were uniformly drawn from the same three sets but the correlation coefficient between arc capacities and interdiction costs was 0.9.

We generated 10 random draws for every topology-capacity-cost combination. Table 3.4.3 shows the average number of arcs for each topology type in our test set. We paired each randomly drawn topology-capacity-cost combination with a total of 100 interdiction budgets: 99 budgets uniformly spaced from 1 percent of the minimum interdiction cost cut to 99 percent of the minimum interdiction cost cut, and one budget equal to 99.99 percent of the minimum interdiction cost cut. As a result, there are a total of 180,000 randomly generated instances.

Each instance was solved using the variable depth flip neighborhood search algorithm, with three different initial solutions:

1. `random` - A random s-t cut.
2. `mincost` - A cut whose arcs have minimum total interdiction cost.
3. `minweightcost` - A cut whose arcs have minimum total weighted interdiction cost; the interdiction costs are weighted by dividing the interdiction costs by the arc capacities.

Initial experiments indicated that the following parameters for the local search provided an effective trade-off between solution time and solution quality: $\mathit{maxSizeAllowed} = 200$, $\mathit{maxDepthAllowed} = 10$, $\mathit{maxBreadthAllowed} = 30$, $\rho = 0.9$, $\lambda = 0.9$, $K = 3$, and $L = 6$. In addition, each instance was allowed up to 10 minutes of computation time with the CPLEX 9.1 callable library. All of the default integer programming settings were used. All computations were performed on an Intel Dual Xeon CPU running at 2.4 Ghz with 1GB of RAM, on the Linux operating system.

3.4.3 Experimental Results

The quality of the solutions produced by our meta-heuristic was very good overall. We compute the quality of a given solution as

$$\text{quality} = \frac{\text{objective value of given solution}}{\text{objective value of optimal solution}}.$$

For the discussion of experimental results in this section, we discard two types of instances.

1. *"Easy" instances.* As a preprocessing step, our algorithm declares an instance "easy" if (i) all the arcs have interdiction cost larger than the interdiction budget, or if (ii) the cost of the minimum interdiction cost cut is less than the interdiction budget. Both these cases are "easy" since they can be detected efficiently as a preprocessing step, and the optimal solution is trivial. Although a number of instances in our test set were detected as "easy" in the first sense, none were detected as "easy" in the second sense, since we only considered instances with interdiction budgets equal to a fraction of the minimum interdiction cost cut. A considerable number of instances in our test set were solved quickly using this pre-processing: 10.95 percent of instances were detected as "easy." Table 3.4.3 shows the fraction of "easy" instances in our test set by network topology type.

2. *Instances for which CPLEX was not able to find an optimal solution in 10 minutes using default settings.* In order to determine the quality of the solutions produced by our meta-heuristic, we solved the instances to optimality using CPLEX. In our initial trials we observed that for some instances CPLEX required several hours to obtain an optimal solution. In the interest of expediency, we terminated CPLEX after ten minutes of running time. This occurred for only 430 instances in our test set, or 0.24 percent. Although CPLEX was terminated early for these instances, the quality of the best feasible solution obtained was comparable to those obtained by our meta-heuristic. Specifically, the average quality for all of these instances was 1.00 for all three possible types of initial solutions. However, for all these instances, our meta-heuristic terminated in under 1 minute.

In Table 3.4.3, we see that when using a minimum interdiction cost cut (`mincost`) as an initial solution, on average, our meta-heuristic produced a solution whose objective value is within 2% of the optimal. This performance is rather uniform across topology types, as shown in Table 3.4.3. Similar results were also achieved by our meta-heuristic when using a minimum weighted interdiction cost cut (`minweightcost`) as an initial solution. In addition, the optimal solution was found by our meta-heuristic using these two initial solution types for more than 90 percent of instances in our test set. These types of initial solutions—`mincost` and `minweightcost`—were much more effective than randomly generated (`random`-type) initial solutions. For example, the quality of the final solutions for instances with network topology types `tpath1` and `tpath5` suffered dramatically when solved using a `random`-type initial solution: in Table 3.4.3, we see that the optimal solution was found for only about 15 percent of instances in our test set with these topology types when using a `random`-type initial solution. On the other hand, the initial solutions of types `mincost` and `minweightcost` were typically already close to optimal for all topology types. This is perhaps a key factor in why our algorithm was able to produce high-quality final solutions when using these types of initial solutions.

Although the quality of the solutions produced by our meta-heuristic was very good, our method usually did not find a solution faster than CPLEX. In Table 3.4.3, we see that even when using initial solutions of type `mincost` or `minweightcost`, our algorithm terminated before CPLEX for only about 14 percent of instances in our test set. On the other hand, for fixed topology, interdiction costs, and capacities, the solution time of our meta-heuristic was insensitive to the value of the interdiction budget. This was not true when CPLEX was used. As we can see in Table 3.4.3 and Table 3.4.3, the variance in solution times is significantly higher for CPLEX than the meta-heuristic for all initial solution and network topology types. Figure 7 shows this phenomenon graphically for a fixed `edge` topology. This behavior was commonly observed, to varying magnitudes, in many different fixed topologies of types `edge`, `path1` and `path5`.

In terms of getting good solutions quickly, our experiments indicate that our meta-heuristic seems to be more effective for dense networks. In our test set, the `arc`-type network topologies were densest; on average, they contained approximately 9,938 arcs, significantly higher than the number of arcs for the other topology types. As seen in Table 3.4.3, the quality of the final solutions produced by our algorithm for instances with `arc`-type topologies is on average superb, regardless of the type of initial solution used. In addition, our meta-heuristic was able to find these high-quality solutions relatively quickly: the results in Table 3.4.3 indicate that on average, our meta-heuristic terminated before CPLEX for instances with `arc`-type network topologies in our experiment, again, regardless of the type of initial solution used; for initial solution types `mincost` and `minweightcost`, our meta-heuristic used about half the time used by CPLEX, on average.

Our conjecture for the observed behavior is as follows: the `edge` topology instances have a large number of edges relative to the number of nodes. The number of edges does not affect our meta-heuristic because our neighborhood is based on moving nodes. The number of crossing arcs in any cut is still small relative to the total number of edges in the network, so the knapsack problems are still relatively small. On the other hand, the denseness of these networks affects CPLEX directly, since there are two variables and one constraint for each edge. More generally, the number of edges profoundly affects the dimension of the problem for CPLEX, but has little effect on the dimension of the subproblems solved during the execution of the meta-heuristic. Furthermore, exploratory experiments also indicate that our algorithm does not scale well for sparse networks. This further confirms our intuition.

We also note that our meta-heuristic is very powerful in the respect that it is unaffected by a poor initial starting solution. This can be observed in Table 3.4.3 where even when an initial solution was chosen at random our meta-heuristic was able to

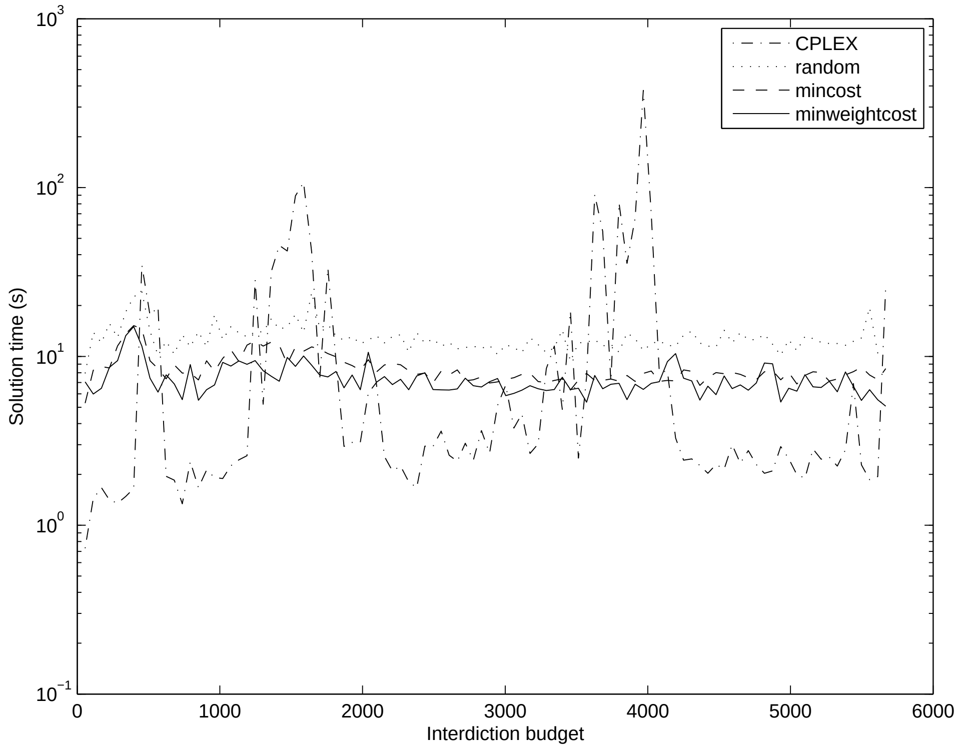

Figure 7: Solution times of an `edge` instance over a wide range of interdiction budgets.

obtain a very good solution for the topologies `edge`, `path1` and `path5`.

Lastly, we observed that the effect of correlation between arc interdiction costs and arc capacities, as well as the effect of correlation between arc interdiction costs and location in network, were both negligible, with respect to both solution quality and speed.

3.5 Conclusions and Future Work

We demonstrate that the time needed to solve Wood's formulation of MFNIP is extremely sensitive to the interdiction budget. This relationship is caused by at least two factors. First, the integrality gap is not bounded by a constant factor. At present, no tight asymptotic bound in terms of the input is known. Second, the

Table 2: Number of arcs by network topology type. Network topology types are defined in Section 3.4.2.

Topology type	Mean number of arcs in the network (standard deviation)
`arc`	9936.90
	(72.45)
`path1`	882.10
	(271.17)
`path5`	2196.70
	(223.11)
`tpath1`	514.70
	(39.46)
`tpath5`	782.40
	(33.90)
All instances	2862.56
	(3588.39)

Table 3: Percentage of "easy" instances by network topology type. "Easy" instances are defined in Section 3.4.3; network topology types are defined in Section 3.4.2.

Topology type	Easy instances (%)
`arc`	0.09
`path1`	9.09
`path5`	2.77
`tpath1`	27.17
`tpath5`	15.63
All instances	10.95

Table 4: Meta-heuristic performance by initial solution type. Initial solution types are defined in Section 3.4.2. Numbers in parentheses are standard deviations. The quality of a given solution is the ratio between the objective value of the solution and the objective value of an optimal solution. The results in this table are limited to instances that were not detected as "easy" by the variable depth neighborhood search algorithm, and instances for which CPLEX was able to find an optimal solution in 10 minutes. "Easy" instances are defined in Section 3.4.3.

Initial solution type	Mean quality of final solution	optimal solution found (%)	Mean solution time (sec)	Meta-heuristc terminates before CPLEX (%)
`random`	1.40	60.77	6.15	8.02
	(0.88)		(5.12)	
`mincost`	1.02	90.06	3.09	14.09
	(0.08)		(2.75)	
`minweightedcost`	1.04	93.84	3.47	13.33
	(0.18)		(3.35)	
CPLEX	1.00	100.0	4.27	N/A
			(37.84)	

Table 5: Mean quality of initial and final meta-heuristic solutions by network topology and initial solution type. Network topology and initial solution types are defined in Section 3.4.2. Numbers in parentheses are standard deviations. The quality of a given solution is defined as the ratio between the objective value of the solution and the objective value of an optimal solution. The results in this table are limited to instances that were not detected as "easy" by the variable depth neighborhood search algorithm, and instances for which CPLEX was able to find an optimal solution in 10 minutes. "Easy" instances are defined in Section 3.4.3.

	`random`		`mincost`		`minweightcost`	
	Initial	Final	Initial	Final	Initial	Final
`arc`	213.651	1.034	1.003	1.001	1.034	1.011
	(532.675)	(0.163)	(0.008)	(0.006)	(0.129)	(0.065)
`path1`	101.916	1.000	1.028	1.000	1.114	1.000
	(89.878)	(0.000)	(0.068)	(0.000)	(0.229)	(0.000)
`path5`	134.981	1.000	1.010	1.000	1.061	1.000
	(180.397)	(0.000)	(0.028)	(0.000)	(0.170)	(0.000)
`tpath1`	121.396	1.905	1.100	1.072	1.340	1.186
	(50.397)	(0.835)	(0.195)	(0.164)	(0.515)	(0.335)
`tpath5`	131.707	2.268	1.049	1.019	1.152	1.042
	(80.585)	(1.399)	(0.127)	(0.087)	(0.363)	(0.208)

Table 6: Percentage of instances in which meta-heuristic finds the optimal solution, by network topology and initial solution type. Network topology and initial solution types are defined in Section 3.4.2. The results in this table are limited to instances that were not detected as "easy" by the variable depth neighborhood search algorithm, and instances for which CPLEX was able to find an optimal solution in 10 minutes. "Easy" instances are defined in Section 3.4.3.

	`random`	`mincost`	`minweightcost`
`arc`	57.94	86.11	71.52
`path1`	100.00	100.00	100.00
`path5`	100.00	100.00	100.00
`tpath1`	15.91	67.00	50.78
`tpath5`	15.33	92.42	90.23

Table 7: Mean solution times by network topology and initial solution type. Network topology and initial solution types are defined in Section 3.4.2. The results in this table are limited to instances that were not detected as "easy" by the variable depth neighborhood search algorithm, and instances for which CPLEX was able to find an optimal solution in 10 minutes. "Easy" instances are defined in Section 3.4.3.

	`random`	`mincost`	`minweightcost`	CPLEX
`arc`	14.17	7.44	8.65	16.93
`path1`	2.86	1.77	1.93	0.30
`path5`	3.72	2.19	2.36	1.45
`tpath1`	4.58	1.71	1.81	0.19
`tpath5`	4.50	1.69	1.79	0.48

Table 8: Standard deviation of solution times by network topology and initial solution type. Network topology and initial solution types are defined in Section 3.4.2. The results in this table are limited to instances that were not detected as "easy" by the variable depth neighborhood search algorithm, and instances for which CPLEX was able to find an optimal solution in 10 minutes. "Easy" instances are defined in Section 3.4.3.

	`random`	`mincost`	`minweightcost`	CPLEX
`arc`	5.12	3.06	3.91	78.63
`path1`	0.32	0.29	0.35	0.75
`path5`	0.69	0.38	0.45	7.03
`tpath1`	2.11	0.31	0.42	0.34
`tpath5`	2.11	0.24	0.31	0.89

number of optimal exposed cuts depends on the interdiction budget. In some cases, there can be exponentially many optimally exposed cuts.

As a remedy to this instability, we developed a robust meta-heuristic that obtains good solutions in a consistent amount of time with respect to the interdiction budget. These results were demonstrated in multiple different classes of randomly generated instances. Moreover, our meta-heuristic was unaffected by the quality of the initial solution, as very good solutions were obtained even when a very poor initial solution happened to be randomly selected. Furthermore, our meta-heuristic did particularly well on dense networks. This is intuitive, as increasing the number of arcs in a network proportionally increases the number of decision variables for Wood's formulation. However, having more arcs in the network does not substantially increase the size of the knapsack subproblems faced by our meta-heuristic, which typically were the bottleneck operation throughout the entire search.

The fact that our neighborhood performs well on dense networks serves as a nice complement to the research on solving instances of MFNIP using Lagrangian relaxation, which includes [67], [11] and [74]. These papers primarily tested instances of MFNIP on grid networks and road networks, both of which are sparse.

The neighborhood we have designed also extends to the multiple resource interdiction variant of MFNIP presented in [77]. The only impediment to the extension is evaluating an exposed cut. In the standard MFNIP, this was a knapsack problem, which is a weakly NP-hard problem that is relatively easy to solve in practice [62]. However, in the case when multiple resources are required for interdiction, this becomes a Multidimensional Knapsack Problem (MDKP), which is strongly NP-hard. However, there is a rich literature on computational approaches to this problem. We leave the extension to multiple resource interdiction for future research. See, for example, [64].

Our present neighborhood search approach suggests that an inner-primal approach might also be appropriate. Specifically, one can view a solution to MFNIP as a partition of the arc set into both interdicted arcs and non-interdicted arcs. Given such an arc partition, the exact inner-primal objective value may be computed with a single maximum flow computation. A variable-depth flip neighborhood that is similar in spirit to Kernighan-Lin search on graph partitioning [49] may be used on the partition of the arc set.

Unfortunately, after we implemented the inner-primal neighborhood, too many maximum flow computations were needed to make this neighborhood viable. However, this motivates an interesting subproblem, The Maximum Flow Reoptimization Problem. This problem may be formally stated as: Given a ground network $N_0 = (V, A_0)$ and a large sequence of sub-networks $N_1 = (V, A_1), N_2 = (V, A_2), \cdots, N_k = (V, A_k)$ that is revealed in an online fashion where $A_i \subseteq A_0 \; \forall \; i \in \{1, 2, \cdots, k\}$. In [57], Nagy and Akl propose the Real Time Maximum Flow Problem, which is similar to spirit in this problem but different in structure. However, no computational results are presented. In the next chapter, we address this problem more rigorously and discuss its applications.

CHAPTER IV

SOLVING ONLINE SEQUENCES OF MAXIMUM FLOWS

In this chapter, we discuss reoptimization heuristics for rapidly solving an online sequence of Maximum Flow Problems (MFPs). In particular, we focus on sequences of MFPs where the ith MFP in the sequence differs from the $(i-1)$st MFP in that exactly one arc has changed, for each possible i. In the first section, we motivate this research and survey related literature. In the second section, we formulate the Maximum Flow Single Arc Reoptimization Problem (MFSAROP) and present a practical algorithm to solve it. In the third section, we present our computational results. In the final section, we draw conclusions and discuss future work.

4.1 Introduction

4.1.1 Motivation

The Maximum Flow Problem (MFP) is a fundamental problem in discrete optimization. Efficient, network algorithms exist to solve instances with thousands of nodes in a matter of seconds. However, despite the existence of large sequences of MFPs in a diverse selection of papers, there does not exist a formalized study of solving a large sequence of MFPs. Given the existence of rapid and scalable algorithms for MFP, it seems intuitive that there is no significant cost to iteratively use a black-box maximum flow solver as a subroutine when solving a sequence of MFPs. Furthermore, effective black-box solvers are easily available on the internet (e.g., [36]). The goal of this chapter, however, is to convince the reader that iteratively using a black-box maximum flow solver to solve a large sequence of MFPs may lead to an enormous number of unnecessary computations.

Sequences of maximum flow problems arise as part of greater computational routines in many, vastly different complex problems. We detail a select, but diverse, set of instances in literature where a sequence of maximum flow computations on topologically similar networks is required.

- *Algorithmic Game Theory* In [26], Devanur et al. introduce a polynomial primal-dual algorithm for computing an equilibrium point for the linear utility case of Fischer markets, which are described in [12]. Their algorithm requires over $O(|V|^4)$ maximum flow computations on a series of bipartite networks with node set V. This equilibrium problem has nontrivial applications in transmission control protocol (TCP) congestion control [48].

- *Bicriteria Network Interdiction* In [77], Royset and Wood numerically compute the Pareto-efficient frontier of the Bi-objective Maximum Flow Network Interdiction Problem. In a maximum flow network interdiction problem, an interdictor allocates a finite amount of resources to remove arcs from a network to minimize the maximum flow in the remaining network. This problem has numerous military applications. To compute the Pareto-efficient frontier for this problem requires a sequence of maximum flow computations.

- *Computational Biology* In [72], Strickland et al. describe an algorithm for estimating the physical similarity between the tertiary structure of two proteins that requires the computations of a quadratic number of maximum matching problems, which are a special case of maximum flow problems.

- *Constraint Programming* In [66], Régin demonstrates how to satisfy constraints of difference, which includes the `all-different` constraint and global cardinality constraints, in Constraint Programming by solving a bipartite matching problem. Thus, to iteratively check if different solutions satisfy a series of `all-different` constraints, this would require a sequence of maximum bipartite matching problems to be solved, which can be modeled as solving a sequence of maximum flow problems.

- *Fingerprint Biometry* Fingerprint matching is a crucial form of universal and reliable identification used by the various law enforcement agencies in the world. According to [40], software for matching a set of fingerprints to an entry in a large database consists of solving a sequence of maximum bipartite matching problems, which can be modeled as solving a sequence of maximum flow problems.

- *Real-Time Process Scheduling* In [71], Stone models real-time scheduling of jobs on a dual-processor computer as an online sequence of minimum s-t capacity cut problems. This is an important problem given the ubiquity of dual-core computing in industry and academia.

- *Robust Network Programming* Robust programming is a method to approach data uncertainty for optimization problems by creating a paradigm for controlling the degree of conservatism of the solution. In Chapter 5, we show that to compute a robust minimum capacity s-t cut (RobuCut), one can solve a sequence of maximum flow problems. RobuCut, has applications to several applications of the Minimum Capacity s-t Cut Problem where arc capacities might be uncertain. For example, those in open-pit mining [43], project scheduling [55] and compiler optimization [78] and [79].

- *Separating Valid Inequalities* In [16], Carr proves that any class of clique-tree inequalities for the Traveling Salesman Problem may be separated by solving a polynomial number of maximum flow problems. For example, a comb inequality with p teeth on a TSP with $|V|$ nodes requires $O(|V|^{2p})$ maximum flow problems to be (implicitly) evaluated.

- *Stochastic Network Programming* Computational research in stochastic network optimization also often requires the evaluation of an expected maximum flow, which requires one maximum flow computation per scenario. In Aneja and Nair [6] and Carey and Hendrickson [15], computing an expected maximum flow is explicitly studied. Computing an expected maximum flow is an integral part of solving stochastic network design problems where the objective is to maximize an expected maximum flow (e.g., Wollmer [76] and Wallace [75]) as well as a stochastic network interdiction problem where the goal is to minimize the expected maximum flow (e.g., Cormican et al. [23]).

We note that in the computational study [20] Cherkassky et al. demonstrate that first-in-first-out and lowest-level-first implementations of the Goldberg-Tarjan algorithm are very competitive with the best augmenting path algorithms for bipartite matching problems. On several instance classes, the results of Cherkassky et al. suggest that the aforementioned Goldberg-Tarjan implementations are faster than the augmenting path algorithms. Thus, even recent studies suggest that using Goldberg-Tarjan algorithms to solve bipartite matching problems is often very practical and therefore, the aforementioned applications that require a sequence of bipartite matching problems to be solved can be viably approached as solving a sequence of MFPs.

In the aforementioned applications, the MFPs are typically topologically similar. That is, the next MFP in the sequence differs from the previous one by adding or removing a small number of arcs or by predictably changing the capacities of a localized arc set. Moreover, when solving these instances, the time and space required to store anything beyond the solution to the previous problem is typically unwarranted. Thus, we model this property by examining *online* sequences of MFPs.

An effective strategy towards quickly solving an entire online sequence of optimization problems is to develop efficient reoptimization heuristics. To this end, we develop a modified maximum flow algorithm that is designed for efficient "warm starts." The motivation behind our choice of algorithm is detailed in the next subsection.

4.1.2 Our Contributions

To allow a study, we formalize the problem at hand into the following:

Maximum Flow Reoptimization Problem (MFROP): Given a ground network $N_0 = (V, A_0)$ and a finite, online sequence of k sub-networks $N_1, N_2, \cdots, N_k$ where $N_i = (V, A_i)$ and $A_i \subseteq A_0 \ \forall \ i \in \{1, \cdots, k\}$, find the maximum flow in each of the sub-networks given that the sequence is revealed in an online fashion.

Since this is an *online* sequence, the ith maximum flow problem must be solved before any knowledge of the $(i + 1)$st maximum flow problem is available beyond that it will be on a sub-network of the ground network.

We will begin the study of solving an entire sequence of maximum flow problems by focusing on a simplified version of MFROP, particularly when N_i and N_{i+1} differ by exactly one arc for all possible i. Formally stated, this problem is as follows:

Maximum Flow Single Arc Reoptimization Problem (MFSAROP): Given a ground network $N_0 = (V, A_0)$ and a finite, online sequence of k sub-networks $N_1, N_2, \cdots, N_k$ where $N_i = (V, A_i)$, $A_i \subseteq A_0 \ \forall \ i \in \{1, \cdots, k\}$ and $|A_{i-1} \oplus A_i| = 1 \ \forall \ i \in \{1, 2, \cdots k\}$, find the maximum flow in each of the sub-networks.

In the problem statement above, $\oplus$ denotes the *symmetric difference* between two sets. That is, $A_{i-1} \oplus A_i = (A_{i-1} \backslash A_i) \cup (A_i \backslash A_{i-1})$. The above problem statement also implicitly includes both allowing an arc's capacity to fluctuate, since parallel arcs can be used, as well as allowing a node to be added or removed, since nodes can be split. This will be detailed later in the manuscript.

Studying the special case of single arc reoptimization is interesting in itself. First, this case is a logical setting to begin our study as it is a simplified version of MFROP. Second, this problem has direct application to real-time scheduling of jobs on a dual-core processor as discussed in [71]. Third, MFSAROP is a subproblem encountered when computing a robust minimum cut, with respect to a polyhedral uncertainty set, using the algorithm detailed in the next chapter.

When designing an algorithm for solving MFSAROP, we modified the Goldberg-Tarjan algorithm. We chose to modify the algorithm of Goldberg and Tarjan for a few reasons. First, this algorithm is considered the fastest algorithm for computing a maximum flow in practice [19] and [53], which suggests that this algorithm could be a good starting point for developing an incremental algorithm. Second, the invariant of the Goldberg-Tarjan algorithm that requires a pre-flow be maintained at each iteration can be easily generalized to require that a pseudo-flow be maintain at each iteration. This is conducive towards efficient reoptimization heuristics after an arc has been deleted. Third, we experimented with iteratively using a Goldberg-Tarjan solver as a black-box versus warm starting CPLEX 9.0's network optimizer for solving a sequence of MFPs that arose during a local search approach to MFNIP. From this experimentation, we observed that the code that evaluated the MFPs using the black-box Goldberg-Tarjan solver ran substantially faster than the code that used the network optimizer as a subroutine.

We did consider designing a reoptimization algorithm based on the maximum flow simplex algorithm in Section 11.8 of [3]. However, we decided not to pursue this further for a few reasons. First, it is well known that network-simplex algorithms tend to have many degenerate pivots, which slow down the algorithm's practical performance. Second, we have empirical confirmation that a network-simplex approach does not work well when arcs are added. We implemented and tested a maximum flow simplex algorithm against our modified Goldberg-Tarjan algorithm for instances of MFROP where the arc capacities were monotonically increasing throughout the sequence. Our modified Goldberg-Tarjan algorithm substantially outperformed the maximum flow simplex solver on these instances. Third, in a network simplex approach to maximum flow reoptimization, if we delete an arc that was in the optimal basis for the previous MFP, there is no immediate way to recover a primal basic feasible solution or a dual basic feasible solution, as defined in the Section 11.9 of [3], without using parallel arcs. Even with the incorporation of parallel arcs, there is no getting around the many degenerate pivots.

We list the contributions of this chapter here. First, we offer an algorithm to solve MFSAROP that exploits information derived from minimum capacity s-t cuts. Second, we demonstrate the significant potential savings from using our algorithm as opposed to using a black-box maximum flow solver for MFSAROP.

4.1.3 Related Work

This section surveys work that is similar in nature to maximum flow reoptimization. In [31], Frangioni and Manca present a computational study of reoptimizing the minimum cost flow problem in the context of decomposition algorithms for a multicommodity minimum cost flow problem. Even though a Maximum Flow Problem (MFP) is a special case of a Minimum Cost Flow Problem (MCFP), to apply reoptimization techniques of MCFP to MFP would fail to exploit the special structure of MFP.

There has also been several papers on the Parametric Maximum Flow Problem (PMFP), which is similar in spirit but very different in problem structure. In PMFP, the objective is to compute the maximum flow in a network where arc capacities are a function of a parameter λ. For examples of papers on PMFP, see [33], [69] and [73].

MFROP is fundamentally different from PFMP. First of all, the sequence of subnetworks in MFROP will be provided in an online fashion. This is certainly different from computing the parametric maximum flow when a finite sequence of desired parameters is known a priori. Secondly, the assumptions on which arcs vary with the parameter is restrictive. Lastly, all of the cited literature on PFMP involves a single parameter, which presents a lot of collinearity in the capacities of the "different" networks that need to be evaluated. This implicit property is also absent in the more general problem of MFROP.

This is not the first publication to recognize the importance of solving a sequence of Maximum Flow Problems. In [57], Nagy and Akl have proposed the Real-Time Maximum Flow Problem (RTMFP), which is essentially the same as MFROP. The only difference is that RTMFP does not involve a ground network. Thus, any number of arcs may be arbitrarily added or deleted. Nagy and Akl introduce RTMFP, discuss a scaling approach for reoptimization and discuss an application to dual-processor scheduling. No computational results are presented.

In [67], Royset and Wood encounter a sequence of maximum flow problems while computing the Pareto-efficient frontier for a bi-objective network interdiction problem. These problems had the special property where the $(i+1)$st network differs from the ith network only in that some of the arcs in the ith network had their capacity increased to form the $(i+1)$st network. As a remedy, the authors implemented a variant of the shortest augmenting path algorithm of Edmonds and Karp [29] that was designed for reoptimization within this context. Specifically, the maximum flow in the ith network was always used as a feasible solution for the $(i+1)$st network.

4.2 The Maximum Flow Single Arc Reoptimization Problem

In this section, we first discuss the complexity of reoptimizing a maximum flow problem in both the circumstance when a new arc is added and when a new arc is removed. Next, we discuss our solution approach to solving MFSAROP, which includes a detailed discussion of our algorithm. Afterwards, we discuss a few extensions of single arc reoptimization. Finally, we propose an enhancement to our algorithm to further reduce the running time.

For a background on complexity theory, we recommend Chapter 15 of Papadimitriou and Steiglitz [60].

4.2.1 Complexity of Reoptimizing a Maximum Flow

We show that the problem of recomputing the maximum flow after a single arc has been added as well as the problem of recomputing the maximum flow after a single arc has been removed are each at least as hard as the Maximum Flow Problem. In addition, we will prove worst-case complexity results on both of these two problems. First, we formally define these two problems.

New Arc Maximum Flow Reoptimization Problem (NAMFRP): Let $N = (V, A)$ be a s-t network where each arc e has a non-negative integer capacity c_e and contains a non-negative flow x_e. Let x^* be a maximum flow in N. Let e' be a new arc that will be added to N to form $N' = (V, A \cup \{e'\})$. Find the maximum flow in N'.

The **Removed Arc Maximum Flow Reoptimization Problem (RAMFRP)** can be defined analogously, where arc e' is removed from N to form the new network $N' = (V, A \backslash \{e'\})$.

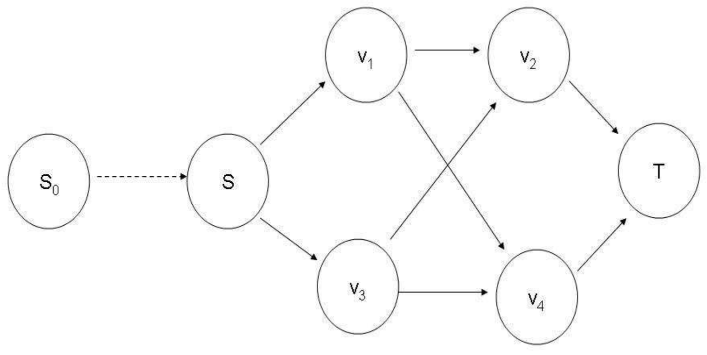

Figure 8: Constructed instance for NAMFRP given a MFP with six nodes.

In each of these problems, we will refer to N as the *previous network*.

4.2.1.1 Hardness Results

Theorem 19. *Recomputing the maximum flow after adding a single arc is P-hard.*

Proof: The proof will be a polynomial reduction from the Maximum Flow Problem (MFP), which is shown to be P-hard in [39].

Consider an arbitrary instance of MFP $\mathcal{I}$ involving network $N = (V, A)$ with source s and sink t. We will define an instance of NAMFRP $\mathcal{I}'$ as follows: Create a new node s_0 and let $V_r = V \cup \{s_0\}$. We will define our previous network as $N_r = (V_r, A)$ with source s_0 and sink t. The maximum flow of our previous network is presently $x^* = 0$ as the source is disconnected. Let the new arc be $e' = (s_0, s)$ and let its capacity be sufficiently large, say $c_{e'} = |V|\ max\ \{c_e \mid e \in A\}$. Clearly the maximum flow of $\mathcal{I}$ is k if and only if the recomputed maximum flow of $\mathcal{I}'$ is k. □

Please see Figure 8 for a sample of this reduction on an acyclic network with six nodes. The dashed arc is the arc that will be added. Given that a constant number of arcs are created to form $\mathcal{I}'$, this is clearly a polynomial reduction.

Theorem 20. *Recomputing the maximum flow after removing a single arc is P-hard.*

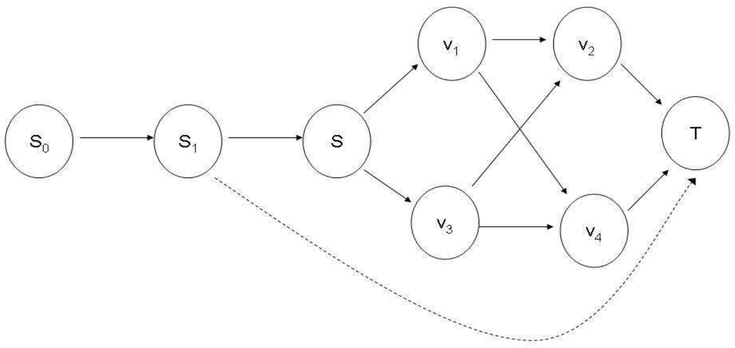

Figure 9: Constructed instance for RAMFRP given a MFP with six nodes.

Proof: The proof will be a log space reduction from the Maximum Flow Problem, which is shown to be P-hard in [39].

Consider an arbitrary instance of MFP $\mathcal{I}$ involving network $N = (V, A)$ with source s and sink t. We will define an instance of RAMFRP $\mathcal{I}'$ as follows: Create two new nodes s_0, s_1 and let $V_r = V \cup \{s_0, s_1\}$. In addition, create three new arcs: $(s_0, s_1), (s_1, s)$ and (s_1, t) and assign each of them a sufficiently large capacity, say $c_M = |V|\ max\ \{c_e \mid e \in A\}$. Let $A_r = A \cup \{(s_0, s_1), (s_1, s), (s_1, t)\}$

We will define our previous network as $N_r = (V_r, A_r)$ with source s_0 and sink t. The maximum flow of the previous network presently takes value c_M. Let x^* be the vector that defines a maximum flow of c_M units along the path $s_0 - s_1 - t$. Let the arc that will be removed be $e' = (s_1, t)$. From inspection, the maximum flow of $\mathcal{I}$ is k if and only if the recomputed maximum flow of $\mathcal{I}'$ is k. $\square$

Please see Figure 9 for a sample of this reduction on an acyclic network with six nodes. The dashed arc is the arc that will be removed. Given that a constant number of arcs are created to form $\mathcal{I}'$, this is clearly a polynomial reduction.

Given both of the reductions above, we note that the worst-case complexity bounds for each of the maximum flow reoptimization cases cannot get any better than solving a new maximum flow problem.

4.2.1.2 Algorithmic Results

The new results in this section use the following algorithmic result of Goldberg and Rao:

Theorem 21. *Consider an instance of the Maximum Flow Problem on a network $N = (V, A)$ where c_e denotes the capacity of arc e for all $e \in A$ and $c_{max} = max\{c_e | e \in A\}$. There exists an algorithm to solve the Maximum Flow Problem in $O(min(|V|^{\frac{2}{3}}, \sqrt{|A|})\ |A|\ log(\frac{|V|^2}{|A|})\ log(c_{max}))$.*

Proof: See Goldberg and Rao [37].

Define $G_r = min(|V|^{\frac{2}{3}}, \sqrt{|A|})\ log(\frac{|V|^2}{|A|})\ log(c_{max})$. We now prove algorithmic results on NAMFRP and RAMFRP.

Theorem 22. *There exists an algorithm for NAMFRP that runs in time $O(min(c_{e'}, G_r)\ |A|)$.*

Proof: This algorithmic result comes from the minimum of two considered algorithms. The first, is to solve the corresponding MFP from scratch, which from Goldberg and Rao we know can be done in $O(G_r|A|)$.

The second considered algorithm is to warm start an augmenting path algorithm using the maximum flow in the network before arc e' was added. An augmenting path in a residual network with m arcs may be found in $O(|A|)$. Since each identified augmenting path allows for at least one more unit of flow to be sent from s to t, at most $c_{e'}$ augmenting paths must be found and thus an instance of NAMFRP may be solved in $O(c_{e'}|A|)$ using this algorithm.

Taking the minimum of the two algorithmic results, we may conclude that there exists an algorithm to solve NAMFRP in $O(min(c_{e'}, G_r)\ |A|)$. □

Theorem 23. *There exists an algorithm for RAMFRP that runs in time $O(min(c_{e'}, G_r)\ |A|)$.*

Proof: Let $e' = (u, v)$ and let $x_{e'}$ be the flow that was on arc e'. We may assume that $x_{e'} > 0$ since otherwise RAMFRP is trivial. Note that when arc e' is removed, node u has a positive excess of $x_{e'}$ and node v has an excess of $-x_{e'}$.

Solving RAMFRP can be viewed as two maximum flow computations. The first is to compute the maximum amount of the $x_{e'}$ units of positive excess flow that can be sent from u to either v or to t in the current residual network. Note that this can be modeled as a MFP by adding a temporary source s', a temporary sink t' and three temporary arcs (s', s), (v, t') and (t, t'), each of which has capacity $x_{e'}$. We note that the maximum flow sent in this problem equals the amount of flow that was on e' that can be "recovered".

The second MFP is only necessary if all of the flow could not be recovered. Suppose that $z' < x_{e'}$ units of flow was recovered in the first MFP. Thus, at this point, node v currently has an excess of $z' - x_{e'} < 0$. We need to push this negative excess into the sink to remove it from the network. This can be achieved by computing a flow of $z' - x_{e'}$ units from t to v, which can be modeled as a MFP.

Note that exactly $x_{e'}$ units of flow is sent in both of the two MFPs combined. As with NAMFRP we can solve each of these MFPs with either the Goldberg-Rao algorithm from scratch or an augmenting path algorithm, whichever provides the better algorithmic result. Thus, there exists an algorithm for RAMFRP that runs in $O(min(c_{e'}, G_r)\ |A|)$. $\square$

Corollary 24. *Given an instance of MFSAROP on a ground network $N = (V, A)$ with an online series of k subnetworks and a maximum capacity of c_{max}, there exists an algorithm to solve this instance in $O(G_r + min(c_{max}, G_r)|A|k)$.*

Proof: The result follows directly from Theorem 22, 23 and the fact that the sequence of subnetworks is of length k. □

4.2.2 Solution Approach

In this subsection, we detail our algorithm for MFSAROP. In subsection 3.2.1, we provide an algorithmic overview. In subsection 3.2.2, we introduce a data structure for storing minimum cuts. Lastly, in subsection 3.2.3, we provide a detailed discussion of our algorithm.

4.2.2.1 Algorithmic Overview

To solve MFSAROP, we implement a modified version of a Goldberg-Tarjan algorithm that is designed for *warm starting*. That is, the capability to start with a good initial solution, which is constructed from the solution of a similar problem. To this end, we will first examine the different types of reoptimization scenarios that may be encountered during the course of solving an online sequence of maximum flow problems. We will then classify all such scenarios into four mutually exclusive and collectively exhaustive scenarios.

Assume that we have already evaluated the $(i-1)$st maximum flow problem, which is on network $N_{i-1} = (V, A_{i-1})$, and let arc $e_i \in A_i \oplus A_{i-1}$ where $N_i = (V, A_i)$ is the network in the ith maximum flow problem. Let c_{e_i} and x_{e_i} be the capacity and flow on arc e_i respectively. If $e_i \notin A_{i-1}$ then we assume $x_{e_i} = 0$. There are two important conditions on arc e_i that are pertinent to efficient reoptimization. First, is the arc added or deleted? Second, is e_i across any of the minimum cuts in N_{i-1}? However, since there could be many minimum cuts in a network, even when the maximum flow is unique, the second question is non-trivial to answer.

Considering the possible answers to these two questions, we introduce four cases for single arc reoptimization. In the interest of brevity, we will simply describe an added or deleted arc e_i that is across a minimum cut in the $(i-1)$st network as being *contained in a minimum cut.*

1. *An added arc e_i is contained in all minimum cuts.* The maximum flow will increase by at least one unit and might increase by at most c_{e_i} units. Calling a modified maximum flow algorithm is necessary in this case.

2. *An added arc e_i is not contained in all minimum cuts.* In this case, the maximum flow in the network will not change. No further computations are needed.

3. *A removed arc $e_i = (u, v)$ was not contained in any minimum cut.* It is possible that a new minimum cut was created. The maximum flow value will decrease by at most x_{e_i} units and will at best be unchanged. Running a modified maximum flow algorithm is necessary in this case.

4. *A removed arc $e_i = (u, v)$ was contained in at least one minimum cut.* The flow will decrease by exactly c_{e_i} units. Since we know that all of the flow on the removed arc cannot be rerouted, we only need to remove the corresponding flow paths. This is significantly easier than running a modified maximum flow algorithm.

We refer two the four cases above as the four *actual reoptimization cases.* The first two cases are considered instances of *new arc reoptimization.* The last two cases are considered instances of *delete arc reoptimization.*

Our algorithm for solving an online sequence of maximum flow problems will consist of iteratively identifying the appropriate reoptimization case and then taking the appropriate action to recompute the maximum flow. The computational details of this will be fleshed out in the rest of this section.

4.2.3 Storing Minimum Capacity s-t Cuts

In this subsection, we discuss a data structure to identify the appropriate case for reoptimization after a single arc has been added or deleted. While recognizing the reoptimization cases 1-3 is not difficult, recognizing the 4th reoptimization case is. There is no clear method to determine if a removed arc is contained in at least one minimum cut that would be faster than performing a maximum flow computation.

In light of this, we have created a data structure that allows us to store two important minimum capacity *s*-*t* cuts:

Definition 3. *Given a network that is currently at maximum flow, a* cut tripartition *is a tripartition* $(V_s, V \backslash (V_s \cup V_t), V_t)$ *of the node set* V *according to the following schema:* V_s *is the set of all nodes currently reachable from the source in the optimal residual network.* V_t *is the set of all nodes that can currently reach the sink in the optimal residual network.*

A cut tripartition implicitly stores two, not necessarily unique, minimum cuts: $C_s = (V_s, V \backslash V_s)$ and $C_t = (V \backslash V_t, V_t)$. Thus, we may alternatively denote a cut tripartition as $\{C_s, C_t\}$.

A cut tripartition can indicate, in constant time, if an arc is contained in all minimum cuts. Specifically, an arc e is contained in all minimum cuts if and only if $e \in C_s \cap C_t$.

A cut tripartition is used later in both Chapters 3 and 4 to obtain a good upper bound on the new maximum flow value in a network after a few arc capacities have been increased. Altner and Ergun first introduced the cut tripartition in [5].

Note that when using a cut tripartition, we cannot catalogue a reoptimization case into one of the four actual reoptimization cases. This is because we cannot determine if a removed arc was in one of the minimum cuts that was not stored. Thus, the four *heuristic reoptimization cases*, which heuristically approximate the actual four cases for reoptimization:

1. Unchanged.

2. Unchanged.

3. *A removed arc $e_i = (u, v)$ was not contained in any **stored** minimum cut.* It is possible that a new minimum cut was created. The maximum flow value will decrease by at most x_{e_i} units and will at best be unchanged. Running a modified maximum flow algorithm is necessary in this case.

4. *A removed arc $e_i = (u, v)$ was contained in at least one **stored** minimum cut.* The flow will decrease by exactly c_{e_i} units. Since we know that all of the flow on the removed arc cannot be rerouted, we only need to remove the corresponding flow paths. This is significantly easier than running a modified maximum flow algorithm.

Our algorithm is still correct if we catalogue our reoptimization scenarios using the heuristic reoptimization cases as opposed to the actual reoptimization cases. The advantage of using the heuristic reoptimization cases is that a case can be identified in constant time when given a properly created cut tripartition. The disadvantage is that when a removed arc is contained in a minimum cut that was not stored, then we will undergo unnecessary computations. Since the remove arc was contained in a minimum cut, the maximum flow value will decrease by the capacity of the removed arc. However, if we are using the heuristic reoptimization scenarios, the only information that will be available is that the removed arc was not in either of the two stored minimum cuts. This misleads the software to attempt to redirect the flow that was on the removed arc through another path, and hence undergo unnecessary computations. To reiterate, we ideally would prefer to use the actual reoptimization cases, but the cost of identifying the actual reoptimization case for removed arcs exceeds the benefit of having the additional information.

4.2.3.1 Reoptimization Algorithm

Since we have not stored all minimum cuts, we evaluate the two deleted arc reoptimization cases by checking if the removed arc is in at least one of the cuts that we have stored, as opposed to all minimum cuts.

When we encounter the ith maximum flow problem, we assume that we have the following information available:

1. The ground network $N_0 = (V, A_0)$.
2. The optimal residual network of the $(i-1)$st network, $N_{i-1} = (V, A_{i-1})$.
3. An arc e_i that will either be added or removed.
4. A cut tripartition built from the $(i-1)$st network.

The main body of the reoptimization algorithm is detailed in Algorithm 4. After the first network N_1 is initialized, we run the Goldberg-Tarjan algorithm to compute the maximum flow in the first network, x_1^*. `GoldbergTarjan(N₁)` is an unmodified Goldberg-Tarjan subroutine, which returns the maximum flow in the parsed network N_1. The next step is to construct a cut tripartition, as defined in Section 3.3.2. This is done using two breadth-first searches in an optimal residual network. The first is to determine all nodes reachable from the source s. The second is to determine all nodes reachable from the sink t. Given a network N with a current flow x, the method `constructCutTripartition(N,x)` constructs a cut tripartition in this fashion.

The next step in Algorithm 4 is to enter a **while** loop. For each new maximum flow problem, there is a new arc that is added (or removed). One of four appropriate subroutines is then selected for reoptimization, depending on which of the four cases detailed above applies. The subroutines for these four heuristic reoptimization cases are detailed in Algorithms 5, 7, 8 and 9 respectively. Given our cut tripartition is already stored, we can determine the applicable heuristic case in constant time. Regardless of which case we are in, we must update the arc set to form the ith network: $N_i = (V, A_i)$.

Algorithm 4 Maximum Flow Reoptimizer Main

```
Initialize network $N_1 \leftarrow (V, A_1)$

$x_1^* \leftarrow$ GoldbergTarjan(N$_1$)

constructCutTripartition(N$_1$,x)

while There is another max flow problem do
  Switch: Heuristic Reoptimization Case
end while
```

The subroutine for the case when a new arc (u, v) is added to all minimum cuts is detailed in Algorithm 5. Given a cut tripartition $(V_s, V\backslash\{V_s, V_t\}, V_t)$, we are in this case if and only if $u \in V_s$ and $v \in V_t$.

Algorithm 5 Case I: Adding a new arc (u, v) across all min. cuts

```
$A_i \leftarrow R(A_{i-1}, x_{i-1}^*) \cup \{(u, v)\}$

$z_i^* \leftarrow z_{i-1}^* +$ modMaxFlow(N$_i$, $x_{i-1}^*$, $c_{(u,v)}$)    // Add pre-flow of $c_{(u,v)}$ units.
```

In this scenario, a new augmenting path is created. The maximum flow will increase by at least one unit and may increase by at most $c_{(u,v)}$ units. A modified maximum flow computation is necessary to exactly determine the new maximum flow value. The subroutine for Case I consists of adding the new arc (u, v) to N_{i-1} to create network N_i and executing a modified maximum flow subroutine, `modMaxFlow(N`$_i$`, x, `Δ_{ub}`)`. This subroutine requires three arguments:

- N_i, the ith network where we must compute a maximum flow.

- x, the current flow that our network will be initialized with.
- Δ_{ub}, the amount of pre-flow that will be added to N_i.

Algorithm 6 `modMaxFlow(N`$_i$`, x, `Δ_{ub}`)`: Modified Goldberg-Tarjan Algorithm

```
$V \leftarrow V \cup \{\bar{s}\}$    // Create a new source $\bar{s}$.
$A \leftarrow A \cup \{(\bar{s}, s)\}$    // Add a new uncapacitated arc.
Initialize $d(v)\ \forall\ v \in V$ using global relabeling
Initialize $e(v)\ \forall\ v \in V \backslash \{\bar{s}\}$ using the existing flow $x$
$e(\bar{s}) \leftarrow \Delta_{ub} \quad x_{(\bar{s},s)} \leftarrow \Delta_{ub}$

while There is an active node $i$ do
  if the residual network contains an admissible arc $(i, j)$ then
    Push $\delta := min\{e(i), c_{(i,j)} - x_{(i,j)}\}$ units of flow from node $i$ to node $j$
  else
    $d(i) \leftarrow \min\ \{d(j) + 1 : (i, j) \in \text{residual } FS(i)\}$
  end if
end while

$V \leftarrow V \backslash \{\bar{s}\}$
$A \leftarrow A \backslash \{(\bar{s}, s)\}$
constructCutTripartition(N,x)
```

The pseudocode for `modMaxFlow(N, x, `Δ_{ub}`)` is contained in Algorithm 6. `modMaxFlow(N, x, `Δ_{ub}`)` is similar to the Goldberg-Tarjan algorithm but there are four key differences. First, it can start from any feasible pre-flow. Second, no additional pre-flow is added to the network, other than Δ_{ub}. This is not mandatory for correctness but instead is intended to be a heuristic improvement. For any reoptimization case, we will have an upper bound z_u on the new maximum flow value. Considering this, it would not be wise to add an amount of pre-flow x_p to the network such that $x_p > z_u$, as we know in advance that $(x_p - z_u)^+$ units of pre-flow would be returned to the source, where $(x)^+ := max\{0, x\}$.

Third, during `modMaxFlow(N, x, `Δ_{ub}`)`, a temporary new source $\bar{s}$ is added to the network and is only incident to the original source s. This is intended to allow the

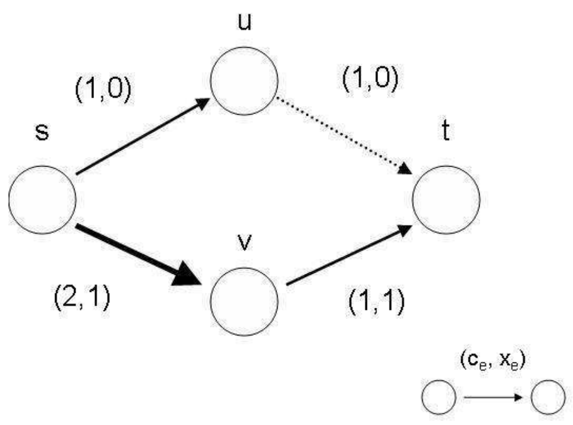

Figure 10: Saturating the "wrong" arc.

original source s to be relabeled, which is not allowed during the standard Goldberg-Tarjan implementation.

We are only adding a bounded amount of pre-flow Δ_{ub} to N_i. In contrast, the original Goldberg-Tarjan algorithm begins by saturating all arcs $FS(s)$. Since we require correctness, we cannot arbitrarily choose which arcs in $FS(s)$ to distribute the pre-flow on. If we were to do so, it is possible that the "wrong" arcs could be saturated. That is, one unit of pre-flow might have been placed on an arc in $FS(s)$ that is not contained in an augmenting path while it could have been placed on a different arc in $FS(s)$ that is contained in at least one augmenting path s-t path, assuming all other flow in the network is unchanged. Here, we use the phrase "augmenting path" to describe a s-t path where all arcs have a non-zero residual capacity.

Figure 10 illustrates a situation where a "wrong" arc is saturated. Assume that the dashed arc (u, t) has just been added to the network. In the diagram, the bold arc (s, v) has been initially saturated when we would prefer to saturate (s, u).

There are pathological examples where it would require less computations to initially saturate all arcs in $FS(s)$ when `modMaxFlow(N, x,` Δ_{ub}`)` is used for new arc reoptimization. However, in practice, it is typically faster to add a bounded amount of pre-flow to the network initially while simultaneously adding a new source to simulate relabeling the source.

The fourth and final difference between `modMaxFlow(N, x,` Δ_{ub}`)` and the original Goldberg-Tarjan Algorithm is that upon termination, `modMaxFlow(N, x,` Δ_{ub}`)` creates a new cut tripartition by executing the method `constructCutTripartition(N,x)`.

Algorithm 7 Case II: Adding a new arc (u, v) that is not in all min. cuts

```
  $A_i \leftarrow R(A_{i-1}, x^*_{i-1}) \ \cup \ \{(u, v)\}$
  $z^*_i \leftarrow z^*_{i-1}$
  newArcTriUpdate(N$_i$, (u,v))    // Update the cut tripartition.
```

The subroutine for the case when a newly added arc (u, v) is not in all minimum cuts is detailed in Algorithm 7. Given a cut tripartition $(V_s, V\backslash\{V_s, V_t\}, V_t)$, we are in this case if and only if either $u \notin V_s$ or $v \notin V_t$.

In this case, we know that the maximum flow value will not change. After adding the new arc, we update our cut tripartition using the method `newArcTriUpdate(N`$_i$`, (u,v))`. This method checks if either node u or node v has become reachable from either s or t. For example, if, without loss of generality, v has become reachable from s and let $R(v)$ be the set of nodes reachable from v in an optimal residual network of N_{i-1} Then we would redefine $V_s \leftarrow V_s \cup \{v\} \cup R(v)$. By presupposition, each node in $R(v)$ was already reachable from v and since v is now reachable from s, all nodes in $R(v)$ are also reachable from s.

The subroutine for the case when an arc (u, v) is deleted that is not in a stored minimum cut is detailed in Algorithm 8. Given a cut tripartition $(V_s, V\backslash\{V_s, V_t\}, V_t)$, we are in this case if and only if the following two booleans are true:

Algorithm 8 Case III: Deleting an arc (u, v) that is not in a stored min. cut

```
$A_i \leftarrow R(A_{i-1}, x^*_{i-1})\backslash\{(u,v)\}$

  // Adding positive and negative excesses to $u, v$ respectively.
$e(u) \leftarrow +x_{(u,v)}$   $e(v) \leftarrow -x_{(u,v)}$

  // Identify $v$ and the sink $t$ to create a new sink $t_v$.
$t_v \leftarrow$ identify(N$_i$, v, t)

$z^*_i \leftarrow z^*_{i-1} - x_{(u,v)} +$ modMaxFlow(N$_i$, $x^*_{i-1}$, 0)

expand(N$_i$, t$_v$)

if $e(v) < 0$ then

    // Remove any remaining ghost flow from the network.
  removeGhostFlow(N$_i$, v, e(v))

end if
```

1. $u \notin V_s$ or $v \notin V\backslash\{V_s, V_t\}$.

2. $u \notin V\backslash\{V_s, V_t\}$ or $v \notin V_t$.

In this case, a modified maximum flow computation is necessary to determine if the flow can be rerouted. The first step is to remove the appropriate arc and add corresponding positive and negative excesses to nodes u and v respectively. Recall that at most $x_{(u,v)}$ units of flow may be lost. To recover this flow, it suffices to either reroute the excess flow, which is now at node u, to either node v or the sink t.

To this end, we temporarily *identify* node v and the sink t. That is, we create a new node t_v where $FS(t_v) = FS(v)$ and $RS(t_v) = RS(v) \cup RS(t)$ and we temporarily remove nodes v and t from the network. This is denoted by the method `identify(N`$_i$`, v, t)`. While v is temporarily removed, we store the negative excess in memory. After this node identification, we set t_v as the new sink. In this context, it is possible for $|FS(t_v)| > 0$, including arcs that both originate and terminate in t_v. Such arcs

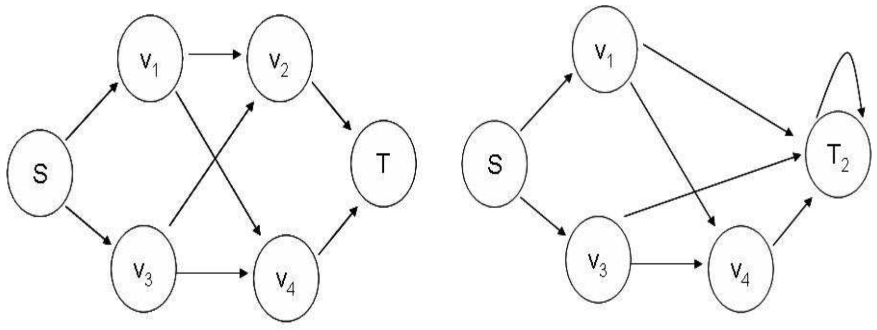

Figure 11: Identifying v_2 and t to form a new sink t_2; before and after.

are often called *loops* in graph theory literature. Please see Figure 11 for an example of node identification.

After v and t are identified, we execute `modMaxFlow(N, x, `Δ_{ub}`)` to determine if all of the excess at node u can be rerouted to the new sink t_v. Note that no additional pre-flow is added to the network when `modMaxFlow(N, x, `Δ_{ub}`)` is called in this situation.

After this subroutine terminates, t_v is expanded back into nodes v and t. This is denoted by the method `expand(N`$_i$`, t`$_v$`)`. If flow was permanently lost through arc deletion or that flow was redirected to the sink then node v will still have a negative excess. This *ghost flow* can be converted into a maximum flow by pushing the negative excess into the sink analogous to how a pre-flow is converted to a maximum flow by pushing the positive excess towards the source. This is denoted by the method `removeGhostFlow(N`$_i$`, v, e(v))`, where $e(v)$ units of ghost flow are pushed from node v to the sink t. This is accomplished using an application of breadth-first search and is detailed in Algorithm 10.

The subroutine for the case when a deleted arc (u, v) is in at least one minimum cut is detailed in Algorithm 9. Given a cut tripartition $(V_s, V\backslash\{V_s, V_t\}, V_t)$, we are in this case if and only if at least one of the following two booleans is true:

Algorithm 9 Case IV: Deleting an arc (u, v) that is in a stored min cut

```
$A_i \leftarrow R(A_{i-1}, x^*_{i-1}) \backslash \{(u, v)\}$

  // $c_{(u,v)}$ units of flow are definitely lost.
$z^*_i \leftarrow z^*_{i-1} - c_{(u,v)}$

  // Remove any remaining pre-flow from the network.
removePreFlow(N$_i$, u, $c_{(u,v)}$)

  // Remove any remaining ghost flow from the network.
removeGhostFlow(N$_i$, v, $c_{(u,v)}$)

  // Update the cut tripartition.
delArcTriUpdate(N$_i$, V, A$_i$)
```

1. $u \in V_s$ and $v \in \{V \backslash V_S, V_t\}$.

2. $u \in \{V \backslash V_S, V_t\}$ and $v \in V_t$.

In this case $c_{(u,v)}$ units of flow is definitely lost. All that is needed in this scenario is to remove the $c_{(u,v)}$ units of pre-flow, remove the $c_{(u,v)}$ units of ghost flow and then update the cut tripartition accordingly. The pre-flow is removed by the method `removePreFlow(N`$_i$`, u, `$c_{(u,v)}$`)`. This method uses breadth-first search and is similar to `removeGhostFlow(N`$_i$`, v, `$c_{(u,v)}$`)`, which removes the ghost flow as previously discussed. `delArcTriUpdate(N`$_i$`, V, A`$_i$`)` checks if any additional nodes are now reachable from either the source or the sink due to the decrease in the maximum flow.

4.2.4 Extensions of Single Arc Reoptimization

This section details extensions of our algorithm for MFSAROP.

Algorithm 10 `removeGhostFlow(N, v,` $e(v)$`)`: Remove $e(v)$ units of Ghost Flow from node v

```
Initialize queue $q \leftarrow \{v\}$
while q is not empty do
    i ← dequeued element from q
    while $e(i) < 0$ do
        Choose $j \in FS(i) : x_{(i,j)} > 0$
        $\Delta \leftarrow \min \{|e(i)|, x_{(i,j)}\}$
        $x_{(i,j)} \leftarrow x_{(i,j)} - \Delta$
        $e(i) \leftarrow e(i) + \Delta$
        Enqueue j in q
    end while
end while
```

4.2.4.1 Changing an Arc Capacity

Our algorithmic framework also implicitly includes both increasing and decreasing a single arc's capacity. To increase the capacity of an arc $e = (u, v)$ by $d_e > 0$, add a parallel arc $\bar{e} = (u, v)$ with capacity $c_{\bar{e}} = d_e$. The modified network may then be reoptimized using Algorithm 4. After reoptimization is complete, the two parallel arcs are *merged* into a single arc.

Definition 4. *Let e_1 and e_2 be two parallel arcs with capacities c_{e_1} and c_{e_2} respectively and flow values x_{e_1} and x_{e_2}. These two arcs are said to be* merged *if we remove arc e_2 from the network and make the following two redefinitions: $x_{e_1} \leftarrow x_{e_1} + x_{e_2}$ and $c_{e_1} \leftarrow c_{e_1} + c_{e_2}$.*

Analogously, to decrease arc e's capacity by d_e, one must first split arc e into two parallel arcs e_1 and e_2 with capacities $c_{e_1} = c_e - d_e$ and $c_{e_2} = d_e$ respectively. If x_e is the original flow on arc e then $x_{e_1} = min\{x_e, c_e - d_e\}$ and $x_{e_2} = max\{0, x_e + d_e - c_e\}$.

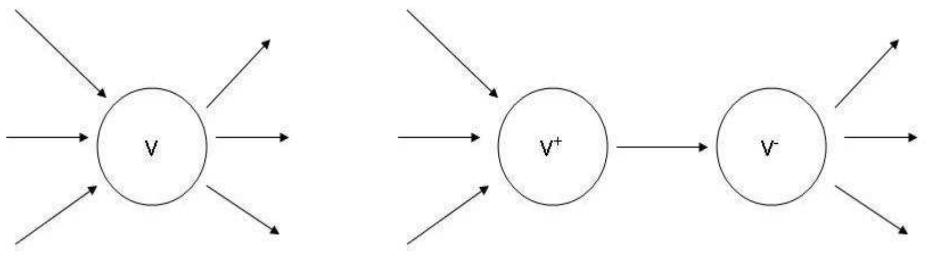

Figure 12: Splitting a Node; Before and After.

One may then remove arc e_2 and reoptimize accordingly.

4.2.4.2 Adding or Deleting a Node

MFSAROP also implicitly includes adding or deleting a single node. However, to do so requires the construction of an auxiliary split-node network $N_a = (V_a, A_a)$. Let $N = (V, A)$ be our original network. N_a will be constructed as follows: for each non-terminal node $v \in V$, that is, a node that is not the source or the sink, we create two nodes $v^+, v^- \in V_a$. For each arc $(u, v) \in A$, there exists an arc $(u^-, v^+) \in A_a$. Moreover, there is a single arc $(v^+, v^-) \in A_a$ with sufficiently large capacity for each node $v \in V$. Adding (removing) node $v \in V$ is equivalent to adding (removing) arc $(v^+, v^-) \in A_a$. N_a will also contain a source s_a and a sink t_a which correspond to the source and sink of N.

4.2.5 Algorithmic Enhancement

In this subsection, we discuss how to accelerate reoptimization after adding a new arc across all minimum cuts. Assume that we have a network that is currently at maximum flow and let (u, v) be an arc that is newly added and is contained in all minimum cuts. Note that if any additional flow can be pushed from s to t, it must be pushed through arc (u, v). Thus, this instance of NAMFRP can be decomposed into the following three step process:

1. Compute the maximum flow from s to u. Call this value z^*_{s-u}.
2. Using at most z^*_{s-u} units of pre-flow, compute the maximum flow from v to t. Call this value z^*_{v-t}.
3. Return $z^*_{s-u} - z^*_{v-t}$ units of flow from u to s.

This procedure will save on distance relabeling computations, as the third step can be accomplished with a breadth-first search, instead of allowing the modified Goldberg-Tarjan algorithm to return flow to the source.

4.3 Computational Results

The purpose of these experiments is to demonstrate the computational savings from using our maximum flow reoptimizer as opposed to using a maximum flow solver as a black-box subroutine. Specifically, the objective of this research is to determine what the best *algorithmic approach* is for solving MFSAROP as opposed to what the best software package is. Thus, we implemented both of the algorithms tested in our experiment to ensure that all other aspects of the coding are equal, such as in the efficiency of the data structures used as well as in efficient memory allocation techniques.

Since both algorithms are just different approaches to computing a maximum flow, there will be no discussion of solution quality, as both methods exactly compute the maximum flow values. Instead, we focus on the reduction in computational time that is achieved from using our reoptimizer.

For a black-box solver, we implemented our own version of the Goldberg-Tarjan algorithm employing both the gap and global relabeling heuristics described in [19]. Our motivation for implementing this ourself is to establish a controlled study. We do not want our algorithms advantages to be obfuscated by processor-specific speed-ups that might exist in a third party software package.

Furthermore, we wish to emphasize that when we ran sequences of maximum flow problems into our black-box solver, we did not deallocate and reallocate memory for our data structures. Memory allocation can be a costly process and we do not intend for the computational savings that stem from reoptimization heuristics to be masked by the additional time required to repeatedly free and construct discrete structures. Instead, after each maximum flow computation, we would empty the data structures to be used for a subsequent computation.

All of these experiments were conducted on a dual Intel Xeon processor each with 2.4 Ghz CPU speed and a cache size of 512 KB. The machine possesses 2.0 GB of RAM.

4.3.1 Maximum Flow Single Arc Reoptimization Problem

Our first set of computational experiments tested the performance of our maximum flow reoptimization algorithm versus a series of iterative calls to a black box solver on randomly generated instances of the Maximum Flow Single Arc Reoptimization Problem (MFSAROP). Each such instance had two input files:

1. A file containing the ground network structure.
2. A file indicating how each network in the sequence differs from the previous network.

We created two classes of problem instances. The first are the `alt` instances and are intended to be relatively dense. These instances consist of a ground network that contains a complete, directed network on the transshipment (non-terminal) nodes along with a single source and a single sink. The arc capacities here are uniformly

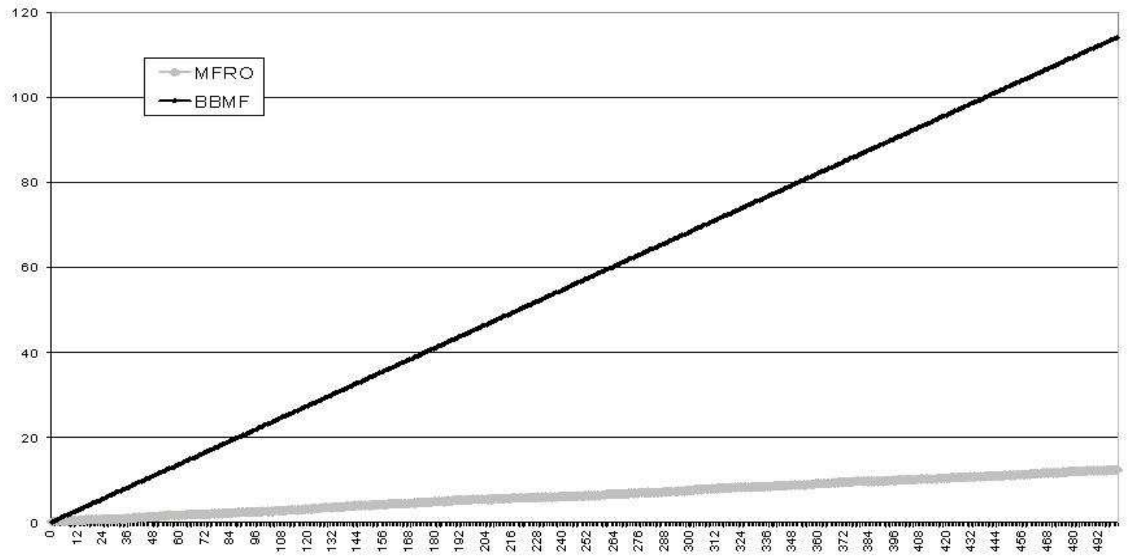

Figure 13: Cumulative time required for each solver to solve a single instance of MFSAROP.

selected from the range [10, 100]. The probability that each arc appears in the first network in the sequence is .7. Then, to construct the sequence of networks, an arc from the ground network $N_0 = (V, A_0)$ is selected uniformly at random to be *flipped* for the next network. That is, suppose we have network $N_i = (V, A_i)$ and we wish to construct $N_{i+1} = (V, A_{i+1})$. Let $e \in A_0$ be the arc that was selected uniformly at random. If $e \in A_i$ then we define $A_{i+1} = A_i \backslash \{e\}$. Analogously, if $e \notin A_i$ then we define $A_{i+1} = A_i \cup \{e\}$.

In our experiment, we chose the number of nodes for a given `alt` instance from the set $\{100, 250, 500, 750, 1000, 2000\}$. We also chose the sequence length from the set $\{100, 200, 300, 400, 500\}$. For each possible pair of number of nodes and sequence lengths, which will henceforth be referred to as a *class*, we generated 5 instances of MFSAROP. The naming convention for the `alt` instances is `alt` followed by a hyphen then the number of nodes followed by another hyphen then the length of the reoptimization sequence. For example, an `alt` instance on 100 nodes with a sequence of length 300 would be named `alt-100-300`.

Figure 13 plots the cumulative time required by the black-box maximum flow algorithm (BBMF) versus our maximum flow reoptimization algorithm (MFRO) on an instance `alt-750-500`. Although the cumulative time required by both algorithms grows linearly with respect to the number of MFPs solved over time, the computational savings from using our reoptimization algorithm becomes more pronounced as the number of reoptimizations required increases.

Table 9 contains the computational results on these instances. The column **Network** contains the class of problem instances. The column **MFBBTime** contains the average number of seconds (over the 5 instances) needed to solve the entire instance of MFSAROP using the maximum flow black-box solver. The column **MFROTime** contains the average number of seconds needed by our maximum flow reoptimizer to solve the entire instance of MFSAROP. The last column, **Perc**, contains the entry in **MFROTime** divided by the entry in **MFBBTime**, written as a percentage.

Clearly we can see that our maximum flow reoptimizer is an order of magnitude faster than the black-box solver. Note that although the time required by our maximum flow reoptimizer does increase relative to the black-box solver as the number of nodes in the network increases, the average time our code requires is less than 20% of the average time required by the black-box solver.

Our second class of problem instances are called the `spa` instances and are intended to be relatively sparse. These instances also consist of a ground network that contains a complete, directed s-t network. The arc capacities here are uniformly selected from the range [10, 100]. The probability that each arc appears in the first network in the sequence is .4. When constructing the sequence of maximum flow problems, given the ith network $N_i = (V, A_i)$ we add an arc to A_i to create A_{i+1} with probability .5 and we remove an arc from A_i in all other situations. Once we decide whether an arc will be added or removed, we then choose an appropriate arc uniformly at random.

Table 9: Computational Results for `alt` Instances

Network	**MFBBTime**	**MFROTime**	**Perc**
alt-100-100	0.308	0.036	11.7%
alt-100-200	0.614	0.066	10.7%
alt-100-300	0.898	0.092	10.2%
alt-100-400	1.226	0.124	10.1%
alt-100-500	1.46	0.158	10.8%
alt-250-100	4.396	0.61	13.9%
alt-250-200	8.668	1.154	13.3%
alt-250-300	12.092	1.744	14.4%
alt-250-400	16.07	2.324	14.5%
alt-250-500	20.102	2.876	14.3%
alt-500-100	16.742	2.54	15.2%
alt-500-200	36.166	4.978	13.8%
alt-500-300	51.34	7.322	14.3%
alt-500-400	68.26	9.668	14.2%
alt-500-500	85.654	12.034	14.0%
alt-750-100	33.196	5.734	17.3%
alt-750-200	65.698	10.962	16.7%
alt-750-300	97.14	16.63	17.1%
alt-750-400	129.534	22.17	17.1%
alt-750-500	170.806	27.916	16.3%
alt-1000-100	65.534	10.294	15.7%
alt-1000-200	123.636	19.706	15.9%
alt-1000-300	153.53	29.276	19.1%
alt-1000-400	204.892	38.858	19.0%
alt-1000-500	298.52	48.674	16.3%
alt-2000-100	277.026	54.722	19.8%
alt-2000-200	592.704	100.734	17.0%
alt-2000-300	945.956	146.612	15.5%
alt-2000-400	1453.22	198.13	13.6%
alt-2000-500	1495.416	253.404	16.9%

We chose the number of nodes for a given spa instance from the set $\{100, 250, 500, 750\}$. We also chose the sequence length from the set $\{100, 200, 300, 400, 500\}$. For each possible pair of number of nodes and sequence lengths, which will henceforth be referred to as a *class*, we generated 9 instances of MFSAROP. The naming convention for the alt instances is alt followed by a hyphen then the number of nodes followed by another hyphen then the length of the reoptimization sequence. For example, an spa instance on 100 nodes with a sequence of length 300 would be named spa-100-300.

Table 10 contains the computational results on the spa instances. The columns are the same as before except the second column is now averaged over 9 instances as opposed to 5. As before, we can see that our maximum flow reoptimizer is an order of magnitude faster than the black-box solver. The average time required by our reoptimization software is consistently under 20% of the time required on average by the black-box solver.

Although there were exceptions, the percentage of black-box solver time required when using the maximum flow reoptimizer is lower for the spa instances as opposed to the alt instances. In randomly generated sparse networks, a removed arc is more likely to be obtained in a known minimum cut and therefore leading to an easier case for removed arc reoptimization. Thus, for the spa instances, we expect there to be less cases of an arc being deleted that is not in a known minimum cut, which is the most time consuming of the four cases for reoptimization.

4.4 Conclusions and Future Work

We demonstrate that the increased time from using a black-box maximum flow solver to solve a large sequence of maximum flow problems can be substantial. As a remedy, we introduced an algorithm designed to solve a large, online sequence of topologically

Table 10: Computational Results for `spa` Instances

Network	**MFBBTime**	**MFROTime**	**Perc**
spa100-100	0.17	0.02	9.2%
spa100-200	0.34	0.03	8.9%
spa100-300	0.50	0.04	8.4%
spa100-400	0.67	0.06	9.0%
spa100-500	0.83	0.07	8.7%
spa250-100	1.73	0.27	15.7%
spa250-200	3.45	0.51	14.9%
spa250-300	5.17	0.76	14.7%
spa250-400	6.88	1.02	14.8%
spa250-500	8.58	1.27	14.8%
spa500-100	8.61	1.21	14.1%
spa500-200	16.95	2.24	13.2%
spa500-300	25.27	3.36	13.3%
spa500-400	35.16	4.48	12.8%
spa500-500	44.12	5.62	12.7%
spa750-100	20.34	2.81	13.8%
spa750-200	40.39	5.34	13.2%
spa750-300	57.73	7.76	13.4%
spa750-400	73.35	10.27	14.0%
spa750-500	91.81	12.68	13.8%

similar maximum flow problems that exploits a simple cut decomposition. Our reoptimization framework typically takes 15% of the time required to solve a randomly generated online sequence of maximum flow problems, where each network differs from the previous network by one arc, when compared to a black-box technique.

One area of possible improvement in our algorithm concerns global relabeling. For MFSAROP, nearly 95% of the computational time is spent on global relabeling, which is done before every call to `modMaxFlow(`N_i`, x, `Δ_{ub}`)` to reset the distance labels. To further reduce the running-time required to solve instances of MFSAROP, we recommend developing heuristics to reduce the time spent resetting all distance labels.

Another area for improvement concerns the reoptimization case where an arc is deleted that is not in any of the two cuts that are stored in the cut tripartition. Since it is possible for a network to have exponentially many minimum cuts, the design of a low maintenance data structure to store all minimum cuts could be of great use, possibly an implementation of the cut decomposition of Picard and Queyranne [61] that would allow the user to determine if an arc is contained in at least one minimum cut in constant time. Such a structure would allow computational savings in Algorithm 4 for MFSAROP. With such a structure, we can use Algorithm 9 for many instances of delete arc reoptimization instead of the slower subroutine Algorithm 8.

We are confident in the potential savings from efficient reoptimization techniques that may be realized in a diverse range of settings, especially those listed in the introduction. We hope this chapter will help advance understanding and spark interest in this area of research.

CHAPTER V

COMPUTING ROBUST MINIMUM CAPACITY S-T CUTS

In this chapter, we extend our maximum flow reoptimization heuristics to rapidly compute robust minimum capacity s-t cuts under a polyhedral model of robustness. In Chapter 4, we study MFSAROP, which has a structure that allows us to use minimum capacity s-t cuts to significantly accelerate the time required to reoptimize a modified MFP. We show that although the Robust Minimum Capacity s-t Cut Problem can be reduced to solving a sequence of MFPs where more than one arc changes between them, there still is a definite structure that can be exploited for problem-specific reoptimization heuristics.

In the first section of this chapter, we motivate this research and discuss relevant literature. In the second section, we formally introduce the Robust Minimum Capacity s-t Cut Problem (RobuCut) and present an algorithmic result. In the third section, we describe our algorithm for RobuCut. In the fourth section, we present computational results. In the last section, we draw conclusions.

5.1 Introduction

The Minimum Capacity s-t Cut Problem (MinCut) is a fundamental problem in combinatorial optimization. It has a plethora of nontrivial applications to a wide selection of real-world problems including, but not limited to, distributed computing on a two processor machine [71], project scheduling [55], open-pit mining [43] and path redundancy elimination during compiler optimization [78] and [79]. For an extensive list of applications, please see Ahuja, Magnanti and Orlin [3].

Given the extensive range of real-world applications, it is natural to study MinCut under data uncertainty. Specifically, we study the problem where arc capacities are unknown but confined to known intervals. This model is useful for the aforementioned applications as the data that corresponds to the arc capacities may be uncertain. For example, in open-pit mining, the economic value of the blocks to be excavated could only be estimated or in distributed computing, the duration of jobs to be scheduled could be uncertain.

Robust programming allows for conservative planning under data uncertainty. Intuitively, robust programming allows a user to maximize his profit or minimize his costs in the worst possible scenario. Since planning for the worst possible scenario is often too conservative, robust programming includes a *parameter of robustness* that allows the decision maker to specify his desired degree of conservative planning. In this chapter, we will use the polyhedral model of uncertainty of Bertsimas and Sim [10].

In [10], Bertsimas and Sim initiated the study of robust combinatorial optimization and network flows. In addition to providing a modeling framework, the authors also proved that any robust combinatorial optimization problem (RobuCOP) can be solved by computing a linear number of nominal combinatorial optimization problems. Thus, a robust minimum capacity s-t cut (RobuCut) may be obtained by solving a linear number of minimum capacity s-t cut problems.

A fundamental result of network optimization is that a minimum capacity s-t cut may be computed by obtaining a maximum s-t flow. Initially, it may seem as if no further work is needed here, as there are quite effective black-box solvers for the maximum flow problem that are easily available. For example, see [19] as well as the corresponding code, which can be found at Goldberg's Network Optimization

Library [36]. However, the purpose of this chapter is to persuade the reader that using a black-box maximum flow solver in this context can lead to a substantial number of unnecessary computations. Sometimes, it might require the overall procedure to take hours when using reoptimization heuristics can reduce the running time to seconds.

RobuCut is also one of several important problems in the burgeoning collection of research literature on robust network programming. We briefly survey a few papers in robust network programming here. In [17], Chaerani and Roos show how to formulate a robust maximum flow problem, using the ellipsoidal model of uncertainty of Ben-Tal and Nemirovski [9], as a conic program. In [8], Atamtürk and Zhang develop a two-stage robust optimization approach for solving network flow and design problems with uncertain demand. The authors generalize the approach to multicommodity flow network and design and given applications to lot-sizing and location-transportation problems. In [59], Ordó*ñ*ez and Zhao develop a robust programming formulation for the problem of expanding arc capacities in a network subject to demand and travel uncertainty. The authors also prove that their model can be reformulated as a conic linear program.

The main contribution of this chapter is providing an efficient algorithm for computing RobuCuts. Specifically, we demonstrate that our algorithm can compute RobuCuts on instances of hundreds of nodes in seconds whereas a naive algorithm that uses a black-box maximum flow solver as a subroutine could take hours on those same instances. Thus, we have turned what would normally take half of a working day into a near real-time decision.

5.2 The Robust Minimum Capacity s-t Cut Problem

In the first subsection in this section, we formally introduce the Robust Minimum Capacity *s-t* Cut Problem. In the second subsection, we present a worst-case algorithmic result.

5.2.1 Problem Statement

Robust Minimum Capacity s-t Cut Problem: Let $N = (V, A)$ be a network with source s and sink t. Assume arc capacities $\tilde{u_e}$ are unknown but are known to take value in $[u_e, u_e + d_e]$ $\forall$ $e \in A$. Choose a minimum capacity *s-t* cut C under the assumption that Γ arc capacities assume capacity $u_e + d_e$, all other arcs have capacity u_e and the Γ arcs are chosen so as to maximize the capacity of C.

We may assume that the arcs are enumerated $A = \{e_0, e_1, \cdots, e_{|A|-1}\}$ such that $d_{e_0} \geq d_{e_1} \geq \cdots \geq d_{e_{|A|-1}}$. For notational convenience, we define $d_{e_{|A|}}$ to be 0. In the robust optimization literature, Γ is referred to as the *robust parameter of optimization.* The user assigns Γ an integer value from the interval $[0, |A|]$. Let ζ be the family of all *s-t* cuts in the network N. The Robust Minimum Capacity *s-t* Cut Problem (RobuCut) can be formally written as follows:

$$\begin{aligned} &Minimize \quad \sum_{e \in C} u_e + max_{\{S | S \subseteq A, |S| \leq \Gamma\}} \sum_{j \in S \cap C} d_j \\ &\qquad Subject\ to \quad C \in \zeta \end{aligned}$$

Theorem 25. *RobuCut may be solved by computing $|A| + 1$ minimum cuts. Specifically, by solving the following optimization problem:*

$$Z^* = min_{\ell=0,\cdots,|A|}\ G^{\ell}$$

where for $\ell = 0, \cdots, |A|$:

$$G^{\ell} := \Gamma d_{e_\ell} + min_{C \in \zeta}\ \{\sum_{e \in C} u_e + \sum_{\{e_j \in C : j \leq \ell\}} (d_{e_j} - d_{e_\ell})\}$$

Proof: Immediate corollary of Theorem 3 in Bertsimas and Sim [10]. □

Corollary 26. *RobuCut may be solved by computing $|A| + 1$ maximum flows.*

Proof: This follows immediately from the previous theorem and by the Maximum Flow Minimum Cut Theorem, which was originally proved in [30]. □

5.2.2 Algorithmic Result

Theorem 27. *Consider an instance of RobuCut on a network $N = (V, A)$. This problem may be solved in $O(G_r + |C^*_{card}|\ |A|\ d_{e_0})$ time where $G_r = min(|V|^{\frac{2}{3}}, \sqrt{|A|})\ |A|\ log(\frac{|V|^2}{|A|})\ log(u_{max})$, C^*_{card} is a minimum cardinality s-t cut in N and $u_{max} = max\{u_e | e \in A\}$.*

Proof: In [37], Goldberg and Rao demonstrated that a maximum flow in a network may be computed in $O(G_r)$, where G_r is defined as above. We may assume that the 0th nominal maximum flow problem is solved by the algorithm of Goldberg and Rao.

Consider the ith nominal maximum flow problem. Note that this problem differs from the $(i-1)$st nominal maximum flow problem in that in the capacities of arcs $e_0, e_1, \cdots, e_i$ is increased by $d_{e_{i-1}} - d_{e_i}$. Let C^*_{card} denote a minimum cardinality *s-t* cut in N. Then the inequality $z^*_i - z^*_{i-1} \leq |C^*_{card}|(d_{e_{i-1}} - d_{e_i})$ holds true, where z^*_i denotes the optimal objective value of the ith nominal maximum flow problem.

Thus, using a maximum flow of the $(i-1)$st nominal maximum flow problem as an initial solution, $|C^*_{card}|(d_{e_{i-1}} - d_{e_i})$ is an upper bound on the maximum number of augmenting paths that must be found in the corresponding residual network until a maximum flow is obtained. Since an augmenting path can be found in at most $O(|A|)$, we conclude that we can compute the maximum flow value of the ith nominal maximum flow problem in $O(|C^*_{card}|(d_{e_{i-1}} - d_{e_i})|A|)$ when we are given a maximum flow in the $(i-1)$st nominal maximum flow problem as a starting solution.

Since we must solve $|A|$ nominal maximum flow problems after the 0th, we see that the total number of computations to compute all of the subsequent maximum flow values may be bounded above by $\sum_{i=1}^{|A|} |C^*_{card}|(d_{e_{i-1}} - d_{e_i})|A| = |C^*_{card}|d_{e_0}|A|$. The desired result follows. $\square$

5.3 Algorithm for RobuCut

This section focuses on our algorithmic approach. In the first subsection, we identify properties of the sequence of maximum flow problems that stem from Corollary 26. In the second subsection, we provide a detailed description of our algorithm. In the third subsection, we prove that our algorithm is no worse, in terms of worst-case analysis, than iteratively using a black-box highest-label Goldberg-Tarjan algorithm $|A| + 1$ times.

5.3.1 Overview of Algorithm

In this subsection, we discuss properties of the sequence of maximum flow problems that must be solved. Recall that in [10], Bertsimas and Sim provide a general algorithm for solving any RobuCOP, which they refer to as Algorithm A. Algorithm A consists of solving a linear number of nominal COPs, which in the case of RobuCut, the sequence of nominal COPs is a sequence of minimum cut problems. To allow for the use of maximum flow algorithms, we take the dual of each of these minimum cut problems.

Let $N_0 = (V, A_0), N_1 = (V, A_1), \cdots, N_{|A|} = (V, A_{|A|})$ be the sequence of *nominal networks*. That is, the networks that underly the $|A| + 1$ maximum flow problems. For each possible i, let u_e^i be the capacity of arc e in network N_i. All of the networks have the same set of arcs although their capacities monotonically increase with i. Let $\bar{A}_i = \{e \in A : u_e^{i-1} < u_i^i\}$. We make the following observations about this sequence of maximum flow problems:

1. $A_i = A_{i+1} \; \forall \; i \in \{0, \cdots, |A| - 1\}$
2. $u_e^i \leq u_e^{i+1} \; \forall \; i \in \{0, \cdots, |A| - 1\}, \; \forall \; e \in A$
3. $\bar{A}_i \subseteq \bar{A}_{i+1} \; \forall \; i \in \{0, \cdots, |A| - 1\}$
4. $|\bar{A_{i+1}} \backslash \bar{A}_i| = 1 \; \forall \; i \in \{0, \cdots, |A| - 1\}$
5. $u_{e_j}^i - u_{e_j}^{i-1} = d_{e_{i-1}} - d_{e_i} \; \forall \; e_j \in \bar{A}_i, \; \forall \; i \in \{1, \cdots, |A|\}$

We derive a heuristic speedup from these properties. Suppose we want to compute the maximum flow in N_i and that we know that x^{i-1} is a maximum flow in N_{i-1}. Note that x^{i-1} is always a feasible flow in N_i for all possible i. Moreover, suppose that there exists an s-t path $P \subseteq \bar{A}_i$. Then we know a priori that the flow of x^{i-1} with $d_{e_{i-1}} - d_{e_i}$ units of flow augmented along path P always routes at least as much flow through N_i as x^{i-1}. Furthermore, we know that the flow of x^{j-1} with $d_{e_{j-1}} - d_{e_j}$ units of flow augmented along path P always routes at least as much flow through N_j as x^{j-1} for each $j \in \{i + 1, i + 2, \cdots |A|\}$. Further still, we can apply the same reasoning to any collection of arc-disjoint s-t paths contained in $\bar{A}_i$.

This suggests the following heuristic: at each iteration we maintain an auxiliary network using the arcs in $\bar{A}_i$ called an *incremental network*. At each iteration $i > 0$, we use the maximum number of arc-disjoint paths in the incremental network along with a maximum flow in N_{i-1} to construct a good feasible flow for N_i. We formally define the incremental network below.

Definition 5. *The* incremental network *for iteration i is the network $\bar{N}_i = (V, \bar{A}_i)$ where all arcs have unit capacity.*

Since $u^i_{e_j} - u^{i-1}_{e_j} = d_{i-1} - d_i \; \forall \; e_j \in \bar{A}_i$, we may assume without loss of generality that every arc in the incremental network has unit capacity. In light of this assumption, each incremental network has a corresponding *multiplier* λ_i where $\lambda_i = d_{i-1} - d_i \; \forall \; i \in \{1, 2, \cdots, |A|\}$.

5.3.2 Algorithm Details

In this subsection, we describe our algorithm for RobuCut in great detail. At the beginning of iteration i, we have the following information stored:

- $N_i = (V, A_i)$, the network where we need to compute a maximum flow.
- x^{i-1}, the maximum flow in the $(i-1)$st network $N_{i-1} = (V, A_{i-1})$.
- $\bar{N}_i = (V, \bar{A}_i)$, the incremental network for the ith iteration along with its corresponding multiplier λ_i.
- $\bar{x}^{i-1}$, the maximum flow in the incremental network $\bar{N}_{i-1} = (V, \bar{A}_{i-1})$.
- A cut tripartition $\{\bar{C}^{i-1}_s, \bar{C}^{i-1}_t\}$ based on an optimal residual network of $\bar{N}_{i-1}$.
- A cut tripartition $\{C^{i-1}_s, C^{i-1}_t\}$ based on an optimal residual network of N_{i-1}.

First we discuss how to use the maximum flow in $\bar{N}_{i-1}$ to compute the maximum flow in $\bar{N}_i$. Second, we discuss how to construct an initial feasible solution for the nominal maximum flow problem on N_i using the maximum flow in $\bar{N}_i$ and the maximum flow in N_{i-1}. Finally, we discuss computing the maximum flow in N_i.

5.3.2.1 *Computing a Maximum Flow in the ith Incremental Network*

We may assume that $i > 0$ since computing a maximum flow in the 0th incremental network is trivial. Let $\{e_{i-1}\} = \bar{A}_i \backslash \bar{A}_{i-1}$. `ReoptIncNetwork(`$\bar{N}_i$`, `$\bar{x}^{i-1}$`, `$e_{i-1}$`, `$\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$`)` is our subroutine for computing a maximum flow in $\bar{N}_i$. Algorithm 11 contains pseudocode for `ReoptIncNetwork(`$\bar{N}_i$`, `$\bar{x}^{i-1}$`, `e_{i-1}`, `$\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$`)`.

Algorithm 11 `ReoptIncNetwork(`$\bar{N}_i$`, `$\bar{x}^{i-1}$`, `e_{i-1}`, `$\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$`)`

```
  $\bar{x}_e^i \leftarrow \bar{x}_e^{i-1} \; \forall \; e \; \in \bar{A}_{i-1}$
  $\bar{x}_{e_{i-1}}^i \leftarrow 0$

  if $e_{i-1} \in \bar{C}_s^{i-1} \cap \bar{C}_t^{i-1}$ then
    findAugmentingPath($\bar{N}_i$, $\bar{x}^i$)
  end if

  $\{\bar{C}_s^i, \bar{C}_t^i\} \leftarrow$ updateCutTripartition($\bar{N}_i$, $\bar{x}^i$, $\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$)

  return ($\bar{x}^i$, $\{\bar{C}_s^i, \bar{C}_t^i\}$)
```

`ReoptIncNetwork(`$\bar{N}_i$`, `$\bar{x}^{i-1}$`, `e_{i-1}`, `$\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$`)` takes four inputs, which are all listed in the parenthesis. This subroutine returns two outputs: a maximum flow $\bar{x}^i$ and a new cut tripartition $\{\bar{C}_s^i, \bar{C}_t^i\}$.

The subroutine `findAugmentingPath(`$\bar{N}_i$`, `$\bar{x}^i$`)` finds an augmenting path in the residual network of $\bar{N}_i$ on flow $\bar{x}^i$ using depth first search. Note that since incremental networks are unit capacity networks, we need to find at most one augmenting path. Lastly, the subroutine `updateCutTripartition(`$\bar{N}_i$`, `$\bar{x}^i$`, `$\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$`)` takes the three inputs listed in the parenthesis and returns a new cut tripartition in the optimal residual network of $\bar{N}_i$. A cut tripartition can always be construct from scratch by using two breadth first search methods, one from the source and the other from the sink. However, when the maximum flow value in $\bar{N}_{i-1}$ equals the maximum flow value in $\bar{N}_i$, it is usually much faster in practice to update the cut tripartition from an optimal residual network of $\bar{N}_{i-1}$ to obtain the cut tripartition from an optimal residual network of $\bar{N}_i$. However, the specific details of how this is done is beyond the scope of this thesis.

5.3.2.2 Constructing a Feasible Flow in the ith Nominal Network

In this section we discuss how we construct a maximum flow in N_i given a maximum flow in an incremental network $\bar{N}_i$ and a maximum flow in the previous network N_{i-1}. To this end, we introduce the following merge operation, which is detailed in Algorithm 12.

Algorithm 12 `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\bar{N}_i$`,` $\bar{x}^i$`,` λ_i`)`

```
for each $e$ in $A_i$ do
  if $e$ is in $\bar{A}_i$ then
    $x^i_e \leftarrow x^{i-1}_e + \lambda_i \bar{x}^i_e$
    $u^i_e \leftarrow u^{i-1}_e + \lambda_i$
  else
    $x^i_e \leftarrow x^{i-1}_e$
    $u^i_e \leftarrow u^{i-1}_e$
  end if
end for

return $x^i$
```

Algorithm 12 contains the pseudocode for the subroutine `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\bar{N}_i$`,` $\bar{x}^i$`,` λ_i`)` where the five inputs for the subroutine are contained within the parenthesis. x^{i-1} denotes a maximum flow in N_{i-1} and $\bar{x}^i$ denotes a maximum flow in $\bar{N}_i$. Thus, x^{i-1}_e and $\bar{x}^i_e$ denote the amount of flow on arc e in flows x^{i-1} and $\bar{x}^i$ respectively. `MergeNetworks(`N_{i-1}`,` x^{i-1}`,` $\bar{N}_i$`,` $\bar{x}^i$`,` λ_i`)` returns x^i a feasible, but not necessarily optimal, flow in N_i.

5.3.2.3 Adding Pre-Flow to the ith Nominal Network

For the purpose of determining how much pre-flow to add to N_i, we obtain a quickly computable upper bound on $z^*_i - z^{init}_i$ where z^*_i denotes the maximum flow value in N_i and z_i^{init} denotes the value of the flow constructed by the subroutine `MergeNetworks(`N_{i-1}`, `x^{i-1}`, `$\bar{N}_i$`, `$\bar{x}^i$`, `λ_i`)`. The impetus for obtaining such an upper bound is that an upper bound indicates how much pre-flow must be added to N_i. More specifically, if $\Delta \geq z_i^* - z_i^{init}$ then each unit of pre-flow added to N_i in excess of Δ units must inevitably be pushed back to the source and most likely result in unnecessary computations. Thus, having a quickly computable upper bound on $z_i^* - z_i^{init}$ allows us to heuristically restrict the amount of pre-flow that we add to N_i.

Let $(V_s^{i-1}, V \backslash (V_s^{i-1} \cup V_t^{i-1}), V_t^{i-1})$ be a cut tripartition on the optimal residual network of N_{i-1} and let $\bar{A}_i^r$ be the set of residual arcs in the optimal residual network of $\bar{N}^i$. Note that after two networks are merged, it is possible for $z_i^* - z_i^{init} > 0$, even if $\{(u,v) \in \bar{A}_i^r : u \in V_s^{i-1}, v \in V_t^{i-1}\} = \emptyset$. Nevertheless, we can compute an upper bound on $z_i^* - z_i^{init}$ using our cut tripartition $(V_s^{i-1}, V \backslash (V_s^{i-1} \cup V_t^{i-1}), V_t^{i-1})$.

Lemma 28. *The following inequality is true:*

$$(d_{e_{i-1}} - d_{e_i})\ \theta_i \geq z_i^* - z_i^{init} \tag{7}$$

where

$$\theta_i = min\ \left\{\ |\{(u,v) \in \bar{A}_i^r : u \in V_s^{i-1}, v \notin V_s^{i-1}\}|,\ |\{(u,v) \in \bar{A}_i^r : v \in V_t^{i-1}, u \notin V_t^{i-1}\}|\ \right\}$$

Proof: z_i^{init} is the objective value of the flow x^i. Moreover, we note that given the feasible flow x^i, the maximum flow in N_i equals z_i^{init} plus the maximum flow in the residual network obtained when x^i is routed through N_i.

Let ζ^r be the set of all s-t cuts in the residual network when x^i is routed through N_i and let r_e^i be the residual capacity of arc e when flow x^i is sent through N_i. From the Maximum Flow Minimum Cut Theorem, we get:

$$z_i^* - z_i^{init} = min_{C^r \in \zeta^r} \sum_{e \in C^r} r_e^i.$$

Let A^r be the set of residual arcs when x^i is routed through N_i. Let $C_s^r = \{(u,v) \in A^r : u \in V_s^{i-1}, v \notin V_s^{i-1}\}$ and let $C_t^r = \{(u,v) \in A^r : u \notin V_t^{i-1}, v \in V_t^{i-1}\}$. Then we obtain the following inequality:

$$z_i^* - z_i^{init} \leq min\{\sum_{e \in C_s^r} r_e^i, \sum_{e \in C_t^r} r_e^i\}.$$

Since both $(V_s^{i-1}, V \backslash V_s^{i-1})$ and $(V \backslash V_t^{i-1}, V_s^{i-1})$ are minimum cuts of N_{i-1} we know that $r_e^i = d_{e_{i-1}} - d_{e_i} \ \forall \ e \in C_s^r \cup C_t^r$ by construction of x^i:

$$z_i^* - z_i^{init} \leq min\{\sum_{e \in C_s^r} d_{e_{i-1}} - d_{e_i}, \sum_{e \in C_t^r} d_{e_{i-1}} - d_{e_i}\},$$

which can be simplified to:

$$z_i^* - z_i^{init} \leq (d_{e_{i-1}} - d_{e_i}) min\{|C_s^r|, |C_t^r|\}.$$

By construction of x^i and from the topological similarities between $\bar{N}_i$ and N_i, we know that $|C_s^r| = |\{(u,v) \in \bar{A}_i^r : u \in V_s^{i-1}, v \notin V_s^{i-1}\}|$ and $|C_t^r| = |\{(u,v) \in \bar{A}_i^r : v \in V_t^{i-1}, u \notin V_t^{i-1}\}|$, which completes the proof. □

5.3.2.4 Computing a Maximum Flow in the ith Nominal Network

We may now formally state our algorithm for RobuCut, whose pseudocode may be found in Algorithm 13.

Algorithm 13 Algorithm for RobuCut

```
$(z_0^*, x^0) \leftarrow$ GoldbergTarjan(N$_0$)
$\{C_s^0, C_t^0\} \leftarrow$ constructCutTripartition($N_0$, $x^0$)
$\bar{x}^0 \leftarrow 0$
$\{\bar{C}_s^0, \bar{C}_t^0\} \leftarrow \{\emptyset, \emptyset\}$

for $i = 1, \cdots, |A|$ do

    if $e_{i-1} \in \bar{C}_s^{i-1} \cap \bar{C}_t^{i-1}$ then

        $(\bar{x}^i, \{\bar{C}_s^i, \bar{C}_t^i\}) \leftarrow$ ReoptIncNetwork($\bar{N}_i$, $\bar{x}^{i-1}$, $e_{i-1}$, $\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$)
    else

        $(\bar{x}^i, \{\bar{C}_s^i, \bar{C}_t^i\}) \leftarrow (\bar{x}^{i-1}, \{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\})$

    end if

    $x^i \leftarrow$ MergeNetworks($N_{i-1}$, $x^{i-1}$, $\bar{N}_i$, $\bar{x}^i$, $d_{e_{i-1}} - d_{e_i}$)

    $\hat{\theta}_i \leftarrow$ ComputeUB($N_i$, $x^i$, $d_{e_{i-1}} - d_{e_i}$, $\{C_s^{i-1}, C_t^{i-1}\}$)

    $z_i^* \leftarrow$ modMaxFlow($N_i$, $x^i$, $\hat{\theta}_i$)

    $\{C_s^i, C_t^i\} \leftarrow$ updateCutTripartition($N_i$, $x^i$, $\{C_s^{i-1}, C_t^{i-1}\}$)

end for

return $\min_{i \in \{0,1,\cdots|A|\}} \Gamma d_{e_i} + z_i^*$
```

Algorithm 13 begins by computing a maximum flow in N_0 using the Goldberg-Tarjan algorithm along with creating a cut tripartition $\{C_s^0, C_t^0\}$ on the optimal residual network of N_0. For notational convenience, we initialize the maximum flow in the 0th incremental network $\bar{x}^0$ to be the trivial flow of 0 units and we initialize two empty cuts for the cut tripartition for the optimal residual network of the 0th incremental network $\{\bar{C}_s^0, \bar{C}_t^0\}$.

The incremental network initially starts with no arcs. Recall that $e_{i-1} \in \bar{A}_i \backslash \bar{A}_{i-1}$. At the beginning of iteration i, the algorithm checks if the cut tripartition corresponding to the $(i-1)$st incremental network, $\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$, indicates that the maximum flow from the in the $(i-1)$st incremental network, $\bar{x}^{i-1}$, is also a maximum flow for the ith incremental network $\bar{N}_i$. If not, then we compute the new maximum flow in $\bar{N}_i$ using the subroutine `ReoptIncNetwork(`$\bar{N}_i$`, `$\bar{x}^{i-1}$`, `e_{i-1}`, `$\{\bar{C}_s^{i-1}, \bar{C}_t^{i-1}\}$`)`. Otherwise, we equate $\bar{x}^i$ and $\{\bar{C}_s^i, \bar{C}_t^i\}$ to their respective values from the previous iteration.

Next we construct an initial feasible flow for N_i using the subroutine `MergeNetworks(`N_{i-1}`, `x^{i-1}`, `$\bar{N}_i$`, `$\bar{x}^i$`, `$d_{e_{i-1}} - d_{e_i}$`)` and we obtain an upper bound on the difference between the maximum flow value in N_i and the current value of flow using the subroutine `ComputeUB(`N_i`, `x^i`, `$d_{e_{i-1}} - d_{e_i}$`, `$\{C_s^{i-1}, C_t^{i-1}\}$`)`, which returns the upper bound proved in Lemma 28.

The penultimate step of an iteration is to compute the maximum flow in N_i using the subroutine `modMaxFlow(`N_i`, `x^i`, `$\hat{\theta}_i$`)`, which is very similar to that of Algorithm 6 discussed in Chapter 4. Finally, we update the cut tripartition for the optimal residual network of N_i using the subroutine `updateCutTripartition(`N_i`, `x^i`, `$\{C_s^{i-1}, C_t^{i-1}\}$`)`. This ends an iteration. After all iterations are complete, the algorithm returns the optimal solution to the RobuCut problem.

Similar to our algorithm for MFSAROP discussed in Chapter 4, we are only adding a bounded amount of pre-flow $\hat{\theta}_i$ to N_i. In contrast, the original Goldberg-Tarjan algorithm begins by saturating all arcs $FS(s)$. Since we arbitrarily choose which arcs in $FS(s)$ to initially distribute $\hat{\theta}_i$ units of pre-flow, it is possible that we could saturate the "wrong" arcs. That is, one unit of pre-flow might have been placed on an arc in $FS(s)$ where that flow cannot possibly reach the source, even if the network currently is not at maximum flow.

Recall that figure 10 in Chapter 4 illustrates a situation where a "wrong" arc is saturated.

5.3.3 Algorithmic Result

In this subsection, we demonstrate that Algorithm 13 is no worse, in terms of worst-case complexity, than solving $|A|$ maximum flow problems from scratch using the highest label implementation of the algorithm of Goldberg and Tarjan.

Theorem 29. *Consider an instance of RobuCut on a network $N = (V, A)$. Algorithm* 13 *runs in time $O(|V|^2|A|^{\frac{3}{2}})$.*

Proof: In [18], Cheriyan and Melhorn proved that the highest label implementation of the Goldberg-Tarjan Algorithm, which is used in our implementation, runs in time $O(|V|^2\sqrt{|A|})$. A modified version of this algorithm, which has the same worst-case algorithmic bound, is called $O(|A|)$ times. This leads to the bound $O(|V|^2|A|^{\frac{3}{2}})$ for all modified maximum flow computations, including solving the 0th nominal maximum flow problem.

What remains to show is a bound on solving the sequence of incremental networks. Recall that to compute the maximum flow value of the ith incremental network given a maximum flow in the $(i-1)$st incremental network as an initial solution requires the computation of at most one augmenting path. Since it takes $O(|A|)$ time to find an augmenting path in a network with $|A|$ arcs, to search for augmenting paths in the sequence of incremental networks takes $1+2+\cdots+|A| = \frac{|A|(|A|-1)}{2} \in O(|A|^2)$. Finally, since for simple networks $O(|A|) \subseteq O(|V|^2)$ this implies $O(|A|^2) \subseteq O(|V|^2|A|) \subseteq O(|V|^2|A|^{\frac{3}{2}})$. The result follows. $\square$

5.4 Computational Results

This section describes our computational experiments, which compared the performance of Algorithm 13 versus a "naive" implementation of the algorithm of Bertsimas and Sim on randomly generated instances of the Robust Minimum Capacity *s*-*t* Cut Problem. The latter algorithm uses a black box maximum flow solver to solve each of the necessary nominal minimum capacity *s*-*t* cut problems from scratch. The Bertsimas and Sim algorithm for RobuCOPs is labeled as Algorithm A in [10]. We specifically implemented this algorithm for computing RobuCuts, which we hereby refer to as the *black-box approach.*

It should be noted that Bertsimas and Sim did not specifically study computing robust minimum capacity *s*-*t* cuts. In addition, we are unaware of any other studies on robust minimum capacity *s*-*t* cuts that uses a polyhedral model of uncertainty.

For the black-box approach, we implemented the following intuitive speedup: During iteration i, if the weight of the incremental network $d_{i-1} - d_i = 0$ then do not recompute the maximum flow, it is the same as in the previous network. This substantially reduces the number of calls to the black box maximum flow solver.

All of the networks we generated are acyclic. When randomly generating a network, we included each possible arc independently with probability p, where p is selected from the set $\{.1, .2, \cdots, .8\}$. We often refer to p as the *arc density parameter.* We chose the number of nodes in the network from the set $\{100, 150, 200, 300, 400, 500\}$. The parameter of robustness Γ was arbitrarily chosen from the appropriate range $[1, |A|]$. We note that both the size and the number of maximum flow computations required to compute a robust minimum capacity s-t cut is independent of Γ. Thus, we believe choosing a single value for Γ is sufficient to test the reduction in computation time when our algorithm is used as opposed to a black-box solver.

Recall how in an instance of RobuCut every arc e has an uncertain capacity $\tilde{u_e}$ that takes value in the range $[u_e, u_e + d_e]$ For each arc e, we selected the value for u_e uniformly at random from the interval [10, 50] and we chose d_e uniformly at random from the interval [5, 20].

For each possible value of the arc density parameter p, for each possible number of nodes and for each possible value of the robust parameter Γ, we generated an instance. The naming convention for these instances is `acyclic-nN-d(100p).net` where N is the number of nodes. For example, an instance on 100 nodes with arc density .5 would be named `acyclic-n100-d50.net`.

Table 11 contains a few select results for our first experiment. The column labeled **File Name** contains the name of the RobuCut instance. The column labeled **MF-BBTime** contains the number of seconds required to solve the instance using our black-box solver. The column labeled **MFROtime** contains the number of seconds required to solve the instance using our reoptimization code for RobuCut. Please note that the computational savings from using our reoptimization algorithm is enormous. In the largest instances in Table 9, the comparison is between over four hours to under thirty seconds. Thus, the advantageousness of algorithm scales well as the number of nodes in the network increases.

5.4.1 Testing the Effectiveness of Incremental Networks

We also conducted an experiment to demonstrate the advantage of using incremental networks during Algorithm 13. To this end, we implemented an algorithm for RobuCut that is essentially the same as Algorithm 13 except that no incremental

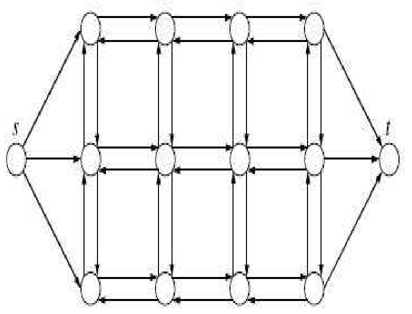

Figure 14: Grid topology with three rows and four columns.

networks are ever used. Thus, in this alternate algorithm, the maximum flow in jth nominal network is computed by using our implementation of the Goldberg-Tarjan algorithm with the bounded pre-flow that uses the maximum flow in the $(j-1)$st nominal network as an initial solution.

5.4.1.1 Instance Generation

For most RobuCut instances on networks on the order of a few hundred nodes, the difference between our algorithm and our algorithm without using incremental networks was not very pronounced. Thus, we limited our comparison of these two algorithms on grid topology networks with at least 100 rows of nodes.

We randomly generate instances of RobuCut on networks with grid topologies. Figure 14 illustrates a sample grid topology with three rows and four columns of nodes. We obtain the idea for these topologies as well as the image from [67].

- The number of rows of nodes is deterministically selected, as a parameter, from the set $\{100, 150, 200, 250\}$. The number of columns of nodes is always twice the number of rows of nodes.
- The lowest possible capacity for each arc e, u_e, is drawn uniformly at random from the interval [10, 50].

- The largest possible increase in an arc's capacity for each arc e, d_e, is drawn uniformly at random from the interval [5, 20].
- The parameter of robustness Γ is arbitrarily chosen to be 20.

For each possible number of rows of nodes r, we generated an instance. The naming convention for these instances is `grid-rx(2r).net`. For example, the class of instances generated with 300 rows of nodes is named `grid-300x600.net`.

Table 12 contains the results of this experiment. The column labeled **File Name** contains the name of the class of RobuCut instances. The columns **MFROtime** and **noINCtime** contain the average number of seconds to solve each of the 10 RobuCut instances for our algorithm and the version of our algorithm without using incremental networks respectively. The column **PerAdv?** contains the percentage of instances for this class where the algorithm using the incremental networks is faster.

These results demonstrate that although using incremental networks is not always superior, it is superior on average. Moreover, these results suggest that the savings that stem from using the incremental networks increases along with the size of the RobuCut instances. This second result is intuitive, since if an s-t path in the incremental is discovered during iteration i, it not only allows one to construct an initial feasible solution for the maximum flow problem on the ith nominal network with a greater objective value than had the incremental network not been used, it also allows us to do the same for each iteration j for all $j > i$. Thus, as the size of the RobuCut instance gets larger, the more iterations required for our algorithm, which in turn means the greater the potential for savings using the incremental network heuristic.

5.5 Conclusions

In this chapter, we studied how to take a conservative approach to the Minimum Capacity *s*-*t* Cut Problem in light of data uncertainty on the arc capacities. Using the polyhedral model of robustness of Bertsimas and Sim, we define the Robust Minimum Capacity *s*-*t* Cut Problem (RobuCut). To this end, we provide a powerful algorithm for computing robust minimum capacity *s*-*t* cuts that builds off of the algorithm of Bertsimas and Sim for general robust combinatorial optimization problems (RobuCOPs). In terms of worst-case complexity, our algorithm is no worse than solving $|A|$ maximum flow problems from scratch.

Furthermore, we demonstrate that our algorithm is very efficient in practice. Our experiments demonstrate the substantial computational savings of maximum flow reoptimization in the context of computing RobuCuts. In particular, our algorithm can solve the largest instances tested in under thirty seconds, while using a black-box solver can take over four hours.

We believe these savings stem from the topological similarities between the $O(|A|)$ maximum flow computations. A lot of information on how the networks differ can be captured by the *incremental network*. We attribute much of the success of our Algorithm 13 to the utilization of the incremental network.

More broadly, we hope that this chapter offers a persuasive argument for employing reoptimization heuristics for even intensively studied and highly structured network flow problems.

Table 11: Computational Results for Robust Minimum Cut

File Name	MFBBTime	MFROtime
acyclic-n150-d20.net	1	0
acyclic-n150-d30.net	1	0
acyclic-n200-d10.net	1	0
acyclic-n100-d60.net	2	0
acyclic-n100-d80.net	2	0
acyclic-n200-d20.net	2	0
acyclic-n300-d10.net	2	0
acyclic-n150-d40.net	3	0
acyclic-n150-d50.net	5	0
acyclic-n200-d30.net	5	0
acyclic-n150-d60.net	9	0
acyclic-n400-d10.net	11	1
acyclic-n150-d70.net	13	0
acyclic-n200-d40.net	14	1
acyclic-n300-d20.net	20	1
acyclic-n150-d80.net	25	1
acyclic-n200-d50.net	32	0
acyclic-n500-d10.net	48	1
acyclic-n200-d60.net	66	1
acyclic-n300-d30.net	89	1
acyclic-n200-d70.net	123	1
acyclic-n400-d20.net	125	1
acyclic-n200-d80.net	183	1
acyclic-n300-d40.net	222	1
acyclic-n300-d50.net	385	3
acyclic-n500-d20.net	407	2
acyclic-n400-d30.net	414	3
acyclic-n300-d60.net	654	3
acyclic-n400-d40.net	882	4
acyclic-n300-d70.net	975	5
acyclic-n500-d30.net	1183	6
acyclic-n300-d80.net	1369	5
acyclic-n400-d50.net	1537	6
acyclic-n500-d40.net	2457	8
acyclic-n400-d60.net	2480	8
acyclic-n400-d70.net	3713	12
acyclic-n500-d50.net	4375	12
acyclic-n400-d80.net	5283	14
acyclic-n500-d60.net	7081	17
acyclic-n500-d70.net	10588	22
acyclic-n500-d80.net	15002	27

Table 12: Advantage of Using Incremental Networks

File Name	**MFROtime**	**noINCtime**	**PerAdv?**
grid-100x200.net	50.4	54.8	100%
grid-150x300.net	265.8	276.4	70%
grid-200x400.net	765.7	833.1	90%
grid-250x500.net	1493.2	1673.8	60%

CHAPTER VI

STOCHASTIC MAXIMUM FLOWS

In this chapter, we discuss extending our maximum flow reoptimization heuristics to rapidly compute a two-stage stochastic maximum flow. The objective of the work summarized in this chapter is to use maximum flow reoptimization heuristics to rapidly compute an expected maximum flow. Unlike the Chapters 4 and 5, there is no special structure to exploit when solving a sequence of MFPs to compute an expected maximum flow if we are not given any overly restrictive information on the distribution for arc failures. Nevertheless, we still can develop useful reoptimization heuristics in light of these new challenges.

In the first section of this chapter, we motivate this work and discuss relevant literature. In the second section, we formally describe the problem as a two-stage stochastic program. In the third section, we present a Benders' reformulation of the problem and design a cutting plane algorithm for its solution. In the fourth section, we present an algorithm to solve a sequence of MFPs when more than one arc changes between each MFP in the sequence. In the fifth section, we describe a variety of tested approaches for incorporating maximum flow reoptimization heuristics into the cutting plane algorithm designed for two-stage stochastic maximum flow. In the final section, we draw conclusions and discuss future work.

6.1 Introduction

The motivation of the research detailed in this chapter is to see if maximum flow reoptimization heuristics can be used to accelerate the computation of a maximum expected maximum flow in the context of two-stage stochastic programming. In many real-world situations, one is compelled to allocate resources towards network design under uncertainty. This chapter investigates how one would commit resources towards expanding the capacities of a network design problem, with the objective of maximizing the expected maximum flow in the network.

We speculate two-stage stochastic maximum flows are applicable in transporting patients to urgent care facilities during an emergency. Consider the following scenario. A central planner, such as the Federal Emergency Management Agency (FEMA), needs to setup multiple urgent care facilities in response to a disaster. There are a finite number of possible locations for these facilities and each potential facility possesses a capacity that corresponds to the number of patients who may be treated at that facility. Furthermore, each urgent care facility has an installation cost and FEMA has a finite budget for installing these facilities.

We assume that the exact number of patients from each location who will need treatment is unknown but can be modeled by a probability mass function. Moreover, due to geographic restrictions, we assume that patients can only be treated at potential urgent care facilities that are within a certain distance of the patients.

The objective of this decision problem is for FEMA to install a subset of urgent care facilities so as to maximize the expected maximum number of patients that can be treated without violating FEMA's budget constraint. Note that this problem can be modeled as a two-stage stochastic maximum flow problem on a bipartite flow network.

There has been a decent number of other papers on stochastic maximum flows. Two-stage stochastic programs with the objective of maximizing the expected maximum flow have been studied in [6], [75] and [76]. In addition, bounds on maximum flows with uncertain arc capacities have been studied in [6], [15], [75] and Chapter 6 of

Kall and Wallace [46].

6.2 Problem Description and Formulation

In this section, we detail the two-stage stochastic maximum flow problem that we investigate.

Two-Stage Stochastic Maximum Flow Problem (2SMFP) Let $N = (V, A)$ be a network with a source s and a sink t. Each arc $e \in A$ has a *potential capacity* $u_e \geq 0$ and a per-unit *investment cost* $r_e \geq 0$ for each unit of the arc's potential capacity that is actually installed. We also assume that the decision maker is given an *investment budget* $B > 0$ for first stage arc investments.

For each arc $e \in A$, let x_e be the fraction of potential capacity u_e that is actually installed during the first stage. Let Ω be a set of known scenarios for arc failures. Given a scenario $\omega \in \Omega$, we say that arc e is *fully operational* under scenario ω if and only if that arc has capacity x_e in the second stage in this scenario. If an arc has a capacity of 0 in the second stage, then we say that the arc is *not operational.* Let ξ_e^ω be a parameter which takes a value of 1 if arc e is fully operational under scenario $\omega \in \Omega$ and takes value 0 if the arc is not operational. We assume that each scenario in Ω is equally likely and that the number of scenarios is a polynomial function of $|V|$ and $|A|$.

The objective of 2SMFP is to invest in arc capacities in the first stage so as to maximize the expected maximum integer flow from s to t subject to the budget constraint. For each arc e, let x_e be the fraction of potential capacity u_e that is actually installed in the first stage. Mathematically, this problem can be written as:

$$max \left\{ \mathbb{E}[F(x, \xi)] : \sum_{e \in A} r_e x_e \leq B, 1 \geq x_e \geq 0 \ \forall \ e \in A \right\} \tag{8}$$

where $\mathbb{E}[F(x,\xi)] = \frac{1}{|\Omega|}\sum_{\omega\in\Omega} F(x,\omega)$ and $F(x,\omega)$ is the maximum flow from s to t in the network where the capacity of arc e is $\xi_e^\omega u_e x_e$.

6.3 Benders' Decomposition Approach

In this section, we first present a reformulation of Problem 8 using Benders' decomposition. We then discuss a second reformulation that aggregates the optimality cuts for the different scenarios. This allows for a cutting plane algorithm that will only add one cut per iteration as opposed to up to $|\Omega|$ cuts per iteration. Our overall approach is modeled after that of Santoso et al. [68].

6.3.1 Benders' Reformulation

Let z_ω denote the maximum flow in the network under scenario ω. We use these variables in Benders' reformulation LP1.

Formulation (LP1)

Maximize $\frac{1}{|\Omega|}\sum_{\omega\in\Omega} z_\omega$

Subject to

$$\sum_{e\in A} r_e x_e \leq B$$

$$z_\omega - \sum_{e\in C} \xi_e^\omega u_e x_e \leq 0 \quad \forall C \in \zeta \quad \forall \omega \in \Omega$$

$$0 \leq x_e \leq 1 \qquad \forall e \in A$$

The objective function is equivalent to maximizing the expected maximum flow since each scenario is equally likely. The first constraint is the budget constraint on the first stage arc investments. The second set of constraints are the *optimality cuts*. In this row, C denotes a s-t cut in the network and ζ denotes the set of all possible s-t cuts in this network. The last set of constraints are the bounds on the x_e variables.

We show how the optimality cuts are obtained in the next lemma.

Lemma 30. *Let $\omega \in \Omega$ be a scenario for an instance of 2SMFP. Then the following is an optimality cut:*

$$z_\omega \leq \sum_{e \in C} \xi_e^\omega u_e x_e. \tag{9}$$

Proof:

Let f_e^ω denote the flow on arc e during scenario ω for each arc $e \in A$. Moreover, let $\hat{x}_e$ be the value assumed by the variable x_e for each arc $e \in A$. We write the second stage subproblem as the following:

$$z^*_\omega(\hat{x}) = max\left\{\sum_{e \in RS(t)} f_e^\omega : \sum_{e \in FS(v)} f_e^\omega - \sum_{e \in RS(v)} f_e^\omega = 0 \ \forall\ v \in V \backslash \{s,t\}; 0 \leq f_e^\omega \leq \xi_e^\omega u_e \hat{x}_e \right\}. \tag{10}$$

Using the widely known Maximum Flow Minimum Cut Theorem [30], we obtain the following equivalence:

$$z^*_\omega(\hat{x}) = min\left\{\sum_{e \in C} \xi_e^\omega u_e \hat{x}_e : C \in \zeta \right\}. \tag{11}$$

This equation suggests the intended upper bound on z_ω. The result follows. $\square$

We note that no feasibility cuts are needed for LP1 since every second stage subproblem is feasible for all feasible values of x. This is because the subproblem for each scenario is a Maximum Flow Problem and the trivial solution of setting the flow on each arc to zero is always a feasible solution to a Maximum Flow Problem when the flow on each arc can take value 0.

6.3.1.1 Cutting Plane Algorithm

Formulation LP1 contains an exponential number of optimality cuts, which are too many to explicitly add to the LP a priori. Thus, we propose solving this LP using a standard constraint generation approach. Algorithm 14 contains pseudocode for a generic cutting plane algorithm for 2SMFP.

Algorithm 14 Generic Cutting Plane Algorithm for LP1

```
$LB \leftarrow 0$
$UB \leftarrow \sum_{e \in FS(s)} u_e$
$P_{mast} \leftarrow \{(x,z) \in \mathcal{R}^{|A|+|\Omega|} : rx \leq B; 0 \leq x \leq 1\}$
$P_{cuts} \leftarrow \{(x,z) \in \mathcal{R}^{|A|+|\Omega|}\}$

repeat
  lpopt($P_{mast} \cap P_{cuts}$)
  $(\hat{x}, \hat{z}) \leftarrow$ getx($P_{mast} \cap P_{cuts}$)
  $UB \leftarrow$ getobjval($P_{mast} \cap P_{cuts}$)
  $LB \leftarrow$ getexpmfval($\hat{x}$)

  if $UB > LB$ then
    $Q_{cuts} \leftarrow$ gencuts($\hat{x}, \hat{z}$)
    $P_{cuts} \leftarrow P_{cuts} \cap Q_{cuts}$
  end if

until $UB == LB$

return $\hat{x}$
```

In Algorithm 14, P_{mast} is the polytope defined by the starting constraints in the *relaxed master problem*; that is, LP1 minus the optimality cuts. P_{cuts} is the polyhedron defined by the intersection of all optimality cuts that have been added to the relaxed master problem. The method `lpopt`(P) maximizes the function $\sum_{\omega \in \Omega} z_\omega$ over the polytope P. An optimal solution and the optimal objective value to the LP solved in `lpopt`(P) are obtained with the methods `getx`(P) and `getobjval`(P) respectively. Given an assignment $\hat{x}$ to the first stage decision variables x, the method `getexpmfval`$(\hat{x})$ returns the expected maximum flow value over all scenarios in Ω. Finally, given $\hat{x}$, the method `gencuts`$(\hat{x})$ returns a polyhedron defined by the intersection of the optimality cut obtained from each second stage subproblem. Let $z^*_\omega(\hat{x})$ denote the value of the maximum flow in the second stage subproblem defined by (11) and let $\hat{z}_\omega$ denote the value that the variable z_ω assumed in the first stage. Then the method `gencuts`$(\hat{x})$ generates an optimality cut (9) for the scenario ω if and only if $z^*_\omega(\hat{x}) < \hat{z}_\omega$.

6.3.2 Aggregating the Optimality Cuts

The disadvantage of the above Benders' reformulation is that a large number of optimality cuts could be added at each iteration during a standard cutting plane algorithm. More constraints means a larger constraint matrix, which requires more running time from any LP-solver that heavily relies on matrix inversion. Since such a solver may need to be iteratively used during Algorithm 14, formulation LP1 may not be desirable.

A potential remedy is to instead aggregate the optimality cuts over all scenarios. Specifically, instead of introducing a decision variable z_ω to correspond to the maximum flow in scenario ω, we will introduce a variable θ to correspond to the value of the expected maximum flow. We can then reformulate (8) as follows:

Formulation (LP2)

Maximize θ

Subject to

$$\sum_{e\in A} r_e x_e \leq B$$

$$\theta - \frac{1}{|\Omega|}\sum_{\omega\in\Omega}\sum_{e\in C^{\omega}} \xi_e^{\omega} u_e x_e \leq 0 \quad \forall C \in \zeta^{|\Omega|}$$

$$0 \leq x_e \leq 1 \quad \forall e \in A$$

In the second row of LP2, C denotes an $|\Omega|$-dimensional vector of s-t cuts in the network and where C^{ω} corresponds to the ωth s-t cut in vector C. We refer to this row as the *aggregated optimality cut.* We omit the proof of correctness of this optimality cut since it is essentially the same as that for the optimality cuts for LP1, which are listed in (9). Furthermore, we omit the generic cutting plane algorithm to solve LP2, since it is essentially the same as Algorithm 14.

6.4 Combining Maximum Flow Reoptimization and the Cutting Plane Method

In this section, we discuss a variety of issues concerning the implementation of Algorithm 14 for solving instances of 2SMFP. In all of our testing, unless otherwise stated, we use formulation LP2. First, we discuss the two possible schemes for maximum flow reoptimization. Second, we discuss the modified Goldberg-Tarjan algorithm that we use for computing maximum flows during Algorithm 14. Third, we discuss reordering the scenarios, as a preprocessing technique, to attempt to reduce the overall time required to solve all of the MFPs using our maximum flow algorithm. Lastly, we discuss our management of the optimality cuts.

6.4.1 Two Schemes for Maximum Flow Reoptimization

Given the structure of Algorithm 14, we note that there are two possible opportunities for usage of maximum flow reoptimization heuristics. The first is reoptimizing within a single iteration of the cutting plane algorithm. That is, using reoptimization heuristics during the subroutine `getexpmfval`($\hat{x}$). This method requires $|\Omega|$ maximum flow problems to be solved. To employ reoptimization heuristics during this subroutine, we use the optimal solution from the ith maximum flow problem to construct a pseudoflow to warm start the $(i+1)$st maximum flow problem. We hereby refer to this scheme for maximum flow reoptimization as the *horizontal reoptimization scheme*. The actual maximum flow algorithm used to compute the new maximum flow is discussed in the next subsection.

The second opportunity for maximum flow reoptimization is between each iteration of the **repeat** loop in Algorithm 14. In this scheme, if we are entering iteration $i > 1$ of the **repeat** loop, then we use the maximum flow from the ωth scenario from the $(i-1)$st iteration to construct a pseudoflow to warm start the maximum flow problem corresponding to the ωth scenario during iteration i. We refer to this scheme as the *vertical reoptimization scheme*.

Unlike the horizontal reoptimization scheme, which only requires that we store the maximum flow from the previous scenario, the vertical reoptimization scheme requires that we store $|\Omega|$ maximum flows, one for each scenario in the previous iteration. This is an obvious disadvantage with respect to storage requirements. However, the potential advantage of the vertical reoptimization scheme is that we might warm start each maximum flow problem with a better solution than in the horizontal reoptimization scheme.

Cutting plane methods are known to wildly oscillate over the feasible region during the early iterations of the algorithm [42]. This suggests that the vertical reoptimization scheme may not be advantageous during the early iterations of the cutting plane algorithm. However, it is possible for the vertical reoptimization scheme to be substantially faster than the horizontal reoptimization scheme in the later iterations of the cutting plane algorithm. Thus, the savings gained at the end of the cutting plane algorithm when the vertical reoptimization scheme is utilized might compensate for the disadvantage incurred during the early iterations.

6.4.2 Modified Goldberg-Tarjan Algorithm

In this subsection, we detail the modified Goldberg-Tarjan algorithm that we used during this study. Our algorithm is a natural extension of Algorithm 6. We note that each consecutive pair of MFPs that arise during Algorithm 14 will almost surely change by more than one arc, so the algorithm developed for the MFSAROP cannot be directly applied.

Suppose that we have found the maximum flow in the ith subnetwork $N_i = (V, A_i)$, which we hereby denote as x^{i*}, and that we want to use this information to compute the maximum flow in the $(i+1)$st subnetwork $N_{i+1} = (V, A_{i+1})$. To this end, we use Algorithm 15 to construct an initial pseudoflow x^{i+1} for the MFP on network N_{i+1}:

The method `addPreflow(`x^{i+1}`)` increases the excess at the source s by a cleverly chosen upper bound Δ_{ub}, where we define Δ_{ub} below. Let C_ℓ^{i*} and C_r^{i*} be the two cuts in the cut tripartition corresponding to an optimal residual network of N_i. Then, by the Maximum Flow Minimum Cut Theorem

$$z_{ub}^{i+1} = min\{\sum_{e\in C_\ell^{i*}} c_e^{i+1}, \sum_{e\in C_r^{i*}} c_e^{i+1}\}$$

is an upper bound on the maximum flow in N_{i+1}. Let z^{i*} denote the value of the

Algorithm 15 `constructPseudoflow(`N_{i+1}`, `x^{i*}`)`: Use x^{i*} to construct a pseudoflow in the network N_{i+1}

```
$e(v) \leftarrow 0 \; \forall \; v \in V$
for $(u, v) \in A_{i+1}$ do
  if $x^{i*} \leq c^{i+1}_{(u,v)}$ then
    $x^{i+1}_{(u,v)} \leftarrow x^{i*}$
  else
    $x^{i+1}_{(u,v)} \leftarrow c^{i+1}_{(u,v)}$
    $e(u) \leftarrow e(u) + x^{i*} - c^{i+1}_{(u,v)}$
    $e(v) \leftarrow e(v) - x^{i*} + c^{i+1}_{(u,v)}$
  end if
end for

$x^{i+1} \leftarrow$ addPreflow($x^{i+1}$)
```

maximum flow in N_i. Thus, considering all of the nodes that already have a positive excess, we need only add the following amount of additional units of pre-flow into the network:

$$\Delta_{ub} = max\{z^{i+1}_{ub} - z^{i*} - \sum_{v \in V : e(v) > 0} e(v), 0\}.$$

For the reasons already articulated in Section 4.2.2.3, we do not wish to add any pre-flow to the source beyond Δ_{ub} as we know that every unit of pre-flow added to the network in excess of Δ_{ub} must eventually be pushed back into the source, and therefore may lead to unnecessary computations.

Let $V^- := \{v \in V : e(v) < 0\}$ and let $dist(u, V^-, N_{i+1}, x^{i+1})$ denote the shortest path, where each arc has a distance of 1 unit, from node u to any of the nodes in V^- in the residual network of when pseudoflow x^{i+1} is assigned in network N_{i+1}. We use Algorithm 16 to start with pseudoflow x^{i+1} and terminate with a maximum flow in N_{i+1}.

The first line of the pseudocode constructs an initial pseudoflow for N_{i+1}, hereby denoted as x^{i+1}, using the method `constructPseudoflow(`N_{i+1}`, `x^{i*}`)`. The second

Algorithm 16 Pseudoflow Goldberg-Tarjan Algorithm

```
$x^{i+1} \leftarrow$ constructPseudoflow($N_{i+1}$, $x^{i*}$)
$d(v) \leftarrow dist(u, V^{-}, N_{i+1}, x^{i+1})\ \forall\ v \in V$
$V \leftarrow V \cup \{\hat{s}\}$
$E \leftarrow E \cup \{(\hat{s}, s)\}$
$x^{i+1}_{(\hat{s},s)} \leftarrow \sum_{e \in FS(s)} x^{i+1}_e$

while There is a node $i \in V \backslash \{\hat{s}\}$ with positive excess do
  if the residual network contains an admissible arc $(i, j)$ then
    Push $\delta := min\{e(i), c^{i+1}_{(i,j)} - x^{i+1}_{(i,j)}\}$ units of flow from node $i$ to node $j$
  else
    $d(i) \leftarrow \min\ \{d(j) + 1 : (i, j) \in$ residual $FS(i)\}$
  end if
end while

$V \leftarrow V \backslash \{\hat{s}\}$
$E \leftarrow E \backslash \{(\hat{s}, s)\}$

while There is a node $i \in V \backslash \{t\}$ with negative excess do
  for $j \in V$ such that $(i, j) \in FS(i)$ do
    Push $\delta := min\{e(i), c^{i+1}_{(i,j)} - x^{i+1}_{(i,j)}\}$ units of flow from node $i$ to node $j$
  end for
end while
```

line of pseudocode initializes all distance labels in the current residual network. In the actual code we used, the distance labels are initialized with the global relabeling heuristic of Cherkassky and Goldberg [19], except where instead of measuring each node's distance to the sink, we temporarily contracted all nodes in V^- into a temporary sink.

The third, fourth and fifth lines create an artificial source $\hat{s}$, so that the original source may be relabeled during the algorithm, for the occasion that flow needs to be redirected through it. Such a circumstance is similar to that discussed in Figure 10.

The first **while** loop in the pseudocode is essentially the unmodified pseudocode of the Goldberg-Tarjan algorithm. However, please note that there might exist nodes with negative excess that may not become active, even if they have flow pushed into them. After the first **while** loop terminates, the temporary source $\hat{s}$ is removed from the network.

The second **while** loop removes all ghost flow from the network. In our actual code, this is done with a breadth-first search routine.

6.4.3 Computational Results

In this subsection, we described our observations when experimenting with our implementation of the cutting plane algorithm listed in Algorithm 14, along with our implementation of the maximum flow solver delineated in Algorithm 16, to solve instances of 2SMFP formulated as stated in LP2.

All computational experiments were conducted on a dual Intel Xeon processor each with 2.4 Ghz CPU speed and a cache size of 512 KB. The machine possesses 2.0 GB of RAM. The Goldberg-Tarjan-based reoptimization algorithm detailed in Algorithm 16 was used to solve all encountered maximum flow problems. All linear programming problems were solved using CPLEX 9.0 under the default settings. All optimality cuts were added to the master LP problem using the method `CPXaddrows()`.

After experimentation, we conclude that using Algorithm 16 to solve the MFPs that are encountered during the course of the cutting plane algorithm does save a little bit of time compared to using a black-box Goldberg-Tarjan maximum flow solver when the horizontal reoptimization scheme was used. The cutting plane algorithm using the vertical reoptimization scheme was usually faster than using a black-box solver for instances that required over a few hundred iterations of the cutting plane algorithm. However, it was not necessarily always faster for instances that required less iterations.

For either reoptimization scheme, the time saved is not substantial. To illustrate, we provide a table of computational results for two prototypical instances.

Table 13: Prototypical Results for Two-Stage Stochastic Maximum Flow

InstanceName	**ReoptScheme**	**TotalTime**	**MaxFlowTime**	**LPTime**
acyclic-n50-p7.net	horizontal	403.82	129.38	269.41
acyclic-n50-p7.net	vertical	404.01	133.47	265.89
acyclic-n50-p7.net	black-box	441.76	161.67	275.28
acyclic-n50-p6.net	horizontal	409.85	131.56	273.54
acyclic-n50-p6.net	vertical	405.59	132.81	268.04
acyclic-n50-p6.net	black-box	440.68	162.21	273.56
acyclic-n40-p6.net	horizontal	113.53	65.25	46.59
acyclic-n40-p6.net	vertical	94.99	57.83	35.71
acyclic-n40-p6.net	black-box	107.84	59.45	46.81
acyclic-n30-p6.net	horizontal	17.68	13.48	3.92
acyclic-n30-p6.net	vertical	27.84	21.14	6.24
acyclic-n30-p6.net	black-box	22.32	18.47	3.58

In Table 13, the column labeled as **InstanceName** contains the name of the instance. The column labeled as **ReoptScheme** indicates whether we used a black-box maximum flow solver, Algorithm 16 in the horizontal reoptimization scheme or Algorithm 16 in the vertical reoptimization scheme. The column labeled as **TotalTime** records the total time, in seconds, for our code to terminate. The columns **MaxFlowTime** and **LPTime** indicate how much of this time was spent on solving MFPs and LPs respectively.

`acyclic-n50-p7.net` is on a randomly generated acyclic network with 50 nodes, roughly 70% of the possible arcs and 100 randomly generated scenarios where each arc has a probability of .9 of failing in any given scenario. The arc capacities are randomly generated integers, drawn uniformly at random, from the interval [1, 50] and the per-unit installation costs are randomly generated, uniformly at random, from the interval [1, 3]. The budget for installing arcs is 150.

`acyclic-n50-p6.net` is similar to `acyclic-n50-p7.net`, except that it has roughly 60% of the possible arcs. `acyclic-n40-p6.net` differs from `acyclic-n50-p7.net` in that it has roughly 60% of the possible arcs, it only has 40 nodes and each arc has the probability of .6 of failing in any given scenario. Similarly, `acyclic-n30-p6.net` differs from `acyclic-n50-p7.net` in that it has roughly 60% of the possible arcs and it only has 30 nodes.

We terminated both `acyclic-n50-p7.net` and `acyclic-n50-p6.net` after 1,000 iterations of the cutting plane algorithm. The algorithm was unable to obtain a provably optimal solution in this amount of time. The algorithm terminated with provably optimal solutions for `acyclic-n40-p6.net` and `acyclic-n30-p6.net` after 971 and 286 iterations respectively.

6.4.3.1 Sorting the Scenarios

This subsection explores the possibility of reducing the running time when the horizontal reoptimization scheme is used by reordering the scenarios.

The horizontal reoptimization scheme currently processes the MFPs in the order that the scenarios are indexed. When using our implementation of Algorithm 16 iteratively to solve a sequence of MFPs, the total running time will be a function of the order in which the MFPs are processed. This suggests that one might be able to reduce the overall running time by re-indexing the scenarios in a sequence that is more conducive to our maximum flow reoptimization algorithm.

Unfortunately, heuristically reordering the scenarios in advance did not reduce the overall computation time required for the cutting plane algorithm. First, we note that optimally reordering the scenarios is essentially as difficult as solving an Asymmetric Traveling Salesman Problem. Specifically, each scenario corresponds to a city and the weight on each arc (i, j) corresponds to the estimated time required to compute the maximum flow under scenario j when warm starting from the maximum flow under scenario i.

There are several explanations for why sorting the scenarios a priori is an effort of questionable advantage. First of all, if we assume that the time to reoptimize between any two scenarios is positively correlated with the Hamming distance between the characteristic vectors of the two scenarios, then to compute the distance matrix for each pair of scenarios requires $O(|A||\Omega|^2)$ computations, which is time consuming.

More importantly, it is doubtful that we have an accurate method to estimate the time required to recompute the maximum flow after transitioning from one scenario to another. Let $F_\omega \subseteq A$ be the set of arcs that are not operational under scenario ω. One metric we tried using to estimate the time to reoptimize from scenario ω_i to ω_j is the Hamming distance of the two characteristic vectors of the scenarios, which equals $|(F_{\omega_i} \backslash F_{\omega_j}) \cup (F_{\omega_j} \backslash F_{\omega_i})|$. We also tried using $|F_{\omega_j} \backslash F_{\omega_i}|$. However, neither allowed us to heuristically reorder the scenarios to reduce the running time of the cutting plane algorithm. Moreover, even if we assume that our reordering heuristic requires 0 seconds to run, the time required for the cutting plane algorithm to terminate on an instance with reordered scenarios is not substantially different from the time required on that same instance with the scenarios in their original order. Simply put, from what we have seen, reordering the scenarios is not a worthwhile pursuit.

6.4.3.2 Management of Optimality Cuts

As the number of iterations of the cutting plane algorithm becomes large, the number of generated optimality cuts increases in turn. A larger constraint matrix increases the time required to solve the relaxed master problem. In many respects, this increased time might be unnecessary, as the optimality cuts added several iterations ago may never become active again.

As a remedy, we heuristically removed optimality cuts that have not been active for an extensive number of iterations from the constraint matrix. Specifically, for each constraint, we maintain a counter for the number of consecutive iterations that the constraint has been inactive at an optimal solution to the relaxed master problem. If this counter exceeds a pre-specified parameter, then this constraint is deleted. Through initial empirical testing, we found this method to be effective at reducing the overall time required to converge to an optimal solution. We also observed that the cutting plane algorithm ran in less time if we deleted constraints as opposed to removing them from the constraint matrix and re-added them as lazy constraints.

6.5 Conclusions and Future Work

We have tried a number of approaches to incorporate maximum flow reoptimization heuristics with the objective of reducing the running time a generic cutting plane algorithm requires for solving a two-stage stochastic maximum flow problem. Although a few of these approaches demonstrated to save a modest amount of time during computational testing, the savings were never substantial.

For future work, we recommend a more intense focus on the best approach towards solving a sequence of MFPs when there is no deterministic structure between the

MFPs that is known a priori. We note that this setting is in contrast to the previous settings explored in this thesis. Both MFSAROP and RobuCut have special structure that can be exploited for developing efficient reoptimization heuristics.

APPENDIX A

PRELIMINARIES

For background in network optimization, we refer the reader to [3]. For a quick overview of linear and integer linear programming, we recommend [58]. For a background in local search, we refer the reader to [1].

A.1 Network Flow Preliminaries

In this thesis, we denote a *network* as $N = (V, A)$ where V is the set of *nodes* and A is the set of *arcs*. We assume without loss of generality that all of our networks have a unique source and a unique sink denoted as s and t respectively, which are *terminals*. All other nodes are *non-terminals*. An arc that originates from node u and terminates in node v is denoted as (u, v). For a node v we denote the set of all arcs entering node v as $FS(v)$ and we denote the set of all arcs leaving node v as $RS(v)$. Given a node v, we denote the in-degree and the out-degree of v as $\delta^-(v)$ and $\delta^+(v)$ respectively.

We refer to a *s-t cut* as either a set of arcs that disconnect s from t upon their removal or alternatively as a bipartition of the nodes that separates s from t. Since the only cuts of interest in this thesis are s-t cuts, we often refer to them as cuts.

Every network mentioned in this thesis is a single commodity flow network and has a unique source $s \in V$ and a unique sink $t \in V$. We assume that there are no arcs entering s and that there are no arcs leaving t. All networks discussed in this thesis are assumed to be s-t connected; that is, there exists a directed path from node s to node t unless otherwise stated. If it is possible to send at least one unit of flow from a node u to a node v then we say that node v is *reachable* from node u.

We refer to a (primal infeasible) solution to a maximum flow problem that obeys the capacity constraints but not the flow balance constraints as a *pseudoflow*.

A.1.1 Maximum Flow Minimum Cut Theorem

The reader should be familiar with the *Maximum Flow Minimum Cut Theorem*, which states that the maximum flow in a s-t network equals the minimum capacity cut. This theorem was originally proved in [30]. For simplicity, we will describe a maximum s-t flow as a maximum flow and a minimum capacity s-t cut as a minimum cut.

A.1.2 Residual Networks

Given a feasible flow x in a network $N = (V, A)$, we can construct the *residual network* as follows. For each node $v \in V$ we create a corresponding node in the residual network. For each arc $e = (u, v) \in A$ such that $c_e - x_e > 0$ we create a corresponding arc $e_f = (u, v)$ in the residual network with capacity $c_{e_f} = c_e - x_e$. Similarly, for each arc $e = (u, v) \in A$ such that $x_e > 0$ we create a corresponding arc $e_b = (v, u)$ in the residual network with capacity $c_{e_b} = x_e$. The source and sink in the residual network correspond to the source and sink respectively of the original network. The following result is well known:

Theorem 31. *A feasible flow is a maximum flow in a network N if and only if the corresponding residual network has a maximum flow of 0.*

A.1.3 Goldberg-Tarjan Algorithm

We refer to the well known maximum flow algorithm of Goldberg and Tarjan [38] as the Goldberg-Tarjan Algorithm. This algorithm is also known as the pre-flow push algorithm or the push-relabel algorithm.

At any point during the Goldberg-Tarjan algorithm, every node v has an associated *distance label* $d(v)$ and an *excess* $e(v)$. The distance label is a lower bound on the shortest distance, in terms of the number of arcs, from v to t. Upon initiation, we set $d(s) = |V|$ and $d(t) = 0$. The excess of a node v is defined as $e(v) = \sum_{i \in RS(v)} x_{(i,v)} - \sum_{j \in FS(v)} x_{(v,j)}$. Any node, that is not the source or sink, with a positive excess is said to be an *active node*.

We say that a residual arc (u, v) is *admissible* if and only if $d(u) = d(v)+1$. Throughout the course of the Goldberg-Tarjan algorithm, admissible arcs are the only arcs that have their current value of flow adjusted.

A *pseudoflow* is a flow that satisfies arc bounds but does not necessarily satisfy the flow balance constraints. A *pre-flow* is a pseudoflow where the flow entering a node is always greater than or equal to the flow leaving a node. We will refer to the quantity $\sum_{i \in V : e(v) > 0} e(v)$ as the *amount of pre-flow* in a network.

The pseudocode for the Goldberg-Tarjan Algorithm is contained in Algorithm 17.

The Goldberg-Tarjan algorithm maintains a pre-flow as an invariant and strives to convert it into a maximum flow. At the beginning of each iteration, we find an active node i. If there is no active node, then we terminate with a maximum flow. Otherwise, we find an admissible arc in $FS(i)$ in the residual network and augment its flow. If no such admissible arc exists, then we *relabel* node i. The step: $d(i) \leftarrow \min \{d(j) + 1 : (i, j) \in \text{residual } FS(i)\}$ denotes relabeling.

Algorithm 17 Goldberg-Tarjan Algorithm

```
Initialize $d(v)$ and $e(v)$ $\forall\ v \in V$
$x_e \leftarrow c_e\ \forall\ e \in FS(s)$
$x_e \leftarrow 0\ \forall\ e \notin FS(s)$

while There is an active node $i$ do
  if the residual network contains an admissible arc $(i, j)$ then
    Push $\delta := min\{e(i), c_{(i,j)} - x_{(i,j)}\}$ units of flow from node $i$ to node $j$
  else
    $d(i) \leftarrow \min\ \{d(j) + 1 : (i, j) \in$ residual $FS(i)\}$
  end if
end while
```

The success of the Goldberg-Tarjan algorithm was partly attributed to the implementation of both gap relabeling and global relabeling heuristics. These heuristics are detailed in [19] and their discussion is beyond the scope of this thesis.

In terms of implementation, we implemented the Highest-Label Goldberg-Tarjan algorithm. That is, we always choose the active node with the highest distance label for the discharge operation. This is regarded as the fastest implementation in practice [19]. We also implemented both the global and the gap relabeling heuristics.

A.2 Integer Programming Preliminaries

Given an instance I of a minimization integer linear program with optimal objective value $z^*_{IP}(I)$ and a corresponding linear programming relaxation with optimal objective value $z^*_{LP}(I)$, we define the $\frac{z^*_{IP}(I)}{z^*_{LP}(I)}$ as the *integrality gap of the instance* I with respect to both the ILP formulation and LP relaxation. We define $sup_{I \in \mathcal{I}} \frac{z^*_{IP}(I)}{z^*_{LP}(I)}$ as the *integrality gap of the problem* with respect to both the ILP formulation and LP relaxation. Here, $\mathcal{I}$ denotes the set of all problem instances.

A.3 Neighborhood Search Preliminaries

A combinatorial optimization problem can generally be denoted as a pair $(\mathcal{S}, f)$ where $\mathcal{S}$ is the set of all feasible solutions and f is the objective function. Typically, a combinatorial optimization problem is stated as $\text{Min}_{s \in \mathcal{S}}\ f(s)$. We define a neighborhood $\mathcal{N}$ as a function $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$, that maps a solution to a set of *neighboring* solutions. Typically, we will just denote the neighborhood of a solution $s \in \mathcal{S}$ by $\mathcal{N}(s)$.

Given a neighborhood, we can construct a *transition graph.* That is, a directed graph where each node corresponds to a solution $s \in \mathcal{S}$ and there exists an arc (i, j) if and only if $j \in \mathcal{N}(i)$.

REFERENCES

[1] E. Aarts and J. K. Lenstra, *Local Search in Combinatorial Optimization*, Princeton University Press, 1997.

[2] R. Ahuja, C. Cunha and G. Sahin, “Network Models in Railroad Planning and Scheduling,” INFORMS Tutorials in Operations Research, 2005.

[3] R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, 1993.

[4] R. Ahuja and J. Orlin, “A Fast and Simple Algorithm for the Maximum Flow Problem,” Operations Research 37, 748-759, 1989.

[5] D. Altner and Ö. Ergun, “Rapidly Solving an Online Sequence of Maximum Flow Problems with Extensions to Computing Robust Minimum Cuts,” *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science 5015, 283-287, 2008.

[6] Y. P. Aneja and K. P. K. Nair, “Maximal Expected Flow in a Network Subject to Arc Failures,” Networks 10, 45-57, 1980.

[7] N. Assimakopoulos, “A network interdiction model for hospital infection control,” Computers in Biology and Medicine 17, 413-422, 1987.

[8] A. Atamtürk and M. Zhang, “Two-Stage Robust Network Flow and Design Under Demand Uncertainty,” Operations Research 55, 662-673, 2007.

[9] A. Ben-Tal and A. Nemirovski, “Robust Convex Optimization,” Mathematics of Operations Research 23, 769-805, 1998.

[10] D. Bertsimas and M. Sim, “Robust Discrete Optimization and Network Flows,” Mathematical Programming 98, 49-71, 2003.

[11] L. Bingol, “A Lagrangian Heuristic for Solving a Network Interdiction Problem,” Master’s Thesis, Naval Postgraduate School, 2001.

[12] W. C. Brainard and H. E. Scarf, “How to Compute Equilibrium Prices in 1891,” Cowles Foundation Discussion Paper, 2000.

[13] G. Brown, M. Carlyle, J. Salmeron and R. K. Wood, “Defending Critical Infrastructure,” Interfaces 36, 530-544, 2006.

[14] C. Burch, R. Carr, S. Krumke, M. Marathe, C. Phillips and E. Sundberg, "A Decomposition-Based Pseudoapproximation Algorithm for Network Flow Inhibition," *Network Interdiction and Stochastic Integer Programming*, Springer US, 51-68, 2003.

[15] M. Carey and C. Hendrickson, "Bounds of Expected Performance of Networks with Links Subject to Failure," Networks 14, 439-456, 1984.

[16] R. Carr, "Separating Clique Trees and Bipartition Inequalities Having a Fixed Number of Handles and Teeth in Polynomial Time," Mathematics of Operations Research 22, 257-265, 1997.

[17] D. Chaerani and C. Roos, "Modelling Some Robust Design Problems via Conic Optimization," Operations Research Proceedings Vol. 2006(VI), 209-214, 2006.

[18] J. Cheriyan and K. Melhorn, "An Analysis of the Highest-Level Selection Rule in the Preflow-Push Max-Flow Algorithm," Information Processing Letters 69, 239-242, 1999.

[19] B. Cherkassky and A. Goldberg, "On Implementing Push-Relabel Method for the Maximum Flow Problem," Algorithmica 19, 390-410, 1994.

[20] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal and J. Stolfi, "Augment or Push: A Computational Study of Bipartite Matching and Unit-Capacity Flow Algorithms," Journal of Experimental Algorithms 3, 1998.

[21] R. Church, M. Scaparra and R. Middleton, "Identifying Critical Infrastructure: The Median and Covering Facility Interdiction Problems," Annals of the Association of American Geographers 94, 491-502, 2004.

[22] W. Cook, W. Cunningham, W. Pulleyblank and A. Schrijver, *Combinatorial Optimization*, Wiley Interscience, 1998.

[23] K. Cormican, D. Morton and R. K. Wood, "Stochastic Network Interdiction," Operations Research 46, 184-197, 1998.

[24] Y. Dai and K. L. Poh, "Solving the Network Interdiction Problem with Genetic Algorithms," Proceedings of the Fourth Asia-Pacific Conference on Industrial Engineering and Management Systems, Taipei, 2002.

[25] H. Derbes, "Efficiently Interdicting a Time-Expanded Transshipment Network," Master's Thesis, Naval Postgraduate School, 1997.

[26] N. Devanur, C. Papadimitrou, A. Saberi and V. Vazirani "Market Equilibrium via a Primal-Dual Algorithm for a Convex Program," Proceedings of the 43rd Annual Symposium on Foundations of Computer Science (FOCS), 2002.

[27] E. A. Dinic, "Algorithm for Solution of a Problem of Maximal Flow in a Network with Power Estimation," Soviet Mathematics Doklady 11, 1277-1280, 1970.

[28] D. Du and R. Chandrasekaran, “The Maximum Residual Flow Problem: NP-hardness with Two-arc Destruction,” Networks, to appear, 2007.

[29] J. Edmonds and R. M. Karp, “Theoretical Improvements in Algorithm Efficiency for Network Flow Problems,” Journal of the ACM 19, 248-264, 1972.

[30] L. R. Ford and D. R. Fulkerson, “Maximal Flow Through a Network,” Canadian Journal of Mathematics 8, 399-404, 1956.

[31] A. Frangioni and A. Manca, “A Computational Study of Cost Reoptimization for Min-Cost Flow Problems,” INFORMS Journal on Computing 18, 61-70, 2006.

[32] S. Fujishige, “A Maximum Flow Algorithm Using MA Orderings,” Operations Research Letters 31, 176-178, 2003.

[33] G. Gallo, M. Grigoriadis and R. Tarjan, “A Fast Parametric Maximum Flow Algorithm and Applications,” SIAM Journal of Computing 18, 30-55, 1989.

[34] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

[35] A. Goldberg, “Recent Developments in Maximum Flow Algorithms,” Scandanavian Workshop of Algorithm Theory (SWAT), 1998.

[36] A. Goldberg, “Andrew Goldberg’s Network Optimization Library,” `http://avglab.com/andrew/soft.html`.

[37] A. Goldberg and S. Rao, “Beyond the Flow Decomposition Barrier,” Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science (FOCS), 2-11, 1997.

[38] A. Goldberg and R. Tarjan, “A New Approach to the Maximum Flow Problem,” Journal of Associated Computing Machinery 35, 1988.

[39] L. M. Goldschlager, R. A. Shaw and J. Staples, “The Maximum Flow Problem is Log Space Complete for P,” Theoretical Computer Science 21, 105-111, 1982.

[40] V. Govindaraju, personal communication with, Professor, Department of Computer Science and Engineering, University at Buffalo, 2008.

[41] T. E. Harris and F. S. Ross, “Fundamentals of a Method for Evaluating Rail Network Capacities,” Research Memorandum RM-1573, The RAND Corporation, 1955.

[42] J. B. Hiriart-Urruty and C. Lemaréchal, *Convex Analysis and Minimization Algorithms II*, Springer-Verlag, Berlin, 1996.

[43] D. Hochbaum and A. Chen, “Performance Analysis and Best Implementations of Old and New Algorithms for the Open-Pit Mining Problem,” Operations Research 48, 894-914, 2000.

[44] E. Israeli and R. K. Wood, “Shortest-path Network Interdiction,” Networks 40, 97-111, 2002.

[45] U. Janjarassuk and J. Linderoth, “Reformulation and Sampling to Solve a Stochastic Network Interdiction Problem”, to appear in Networks, 2007.

[46] P. Kall and S. Wallace, *Stochastic Programming*, John Wiley and Sons, Inc., New York, 1994.

[47] A. V. Karzanov, “Determining the Maximal Flow in a Network with the Method of Preflows,” Soviet Mathematics Doklady 15, 434-437, 1974.

[48] F. P. Kelly, “Charging and Rate Control for Elastic Traffic,” European Transactions on Telecommunications 8, 33-37, 1997.

[49] B. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” Bell System Technical Journal 49, 291-307, 1970.

[50] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, Volume 21 of Algorithms and Combinatorics, Springer, Berlin, Second Edition, 2002.

[51] C. Lim and J. C. Smith, “Algorithms for Discrete and Continuous Multicommodity Flow Network Interdiction Problems,”IIE Transactions 39, 15-26, 2007.

[52] V. M. Malhotra, M. P. Kumar and S. N. Maheshwari, “An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks,” Information Processing Letters 7, 277-278, 1978.

[53] Y. Matsuoka and S. Fujishige, “Practical Efficiency of Maximum Flow Algorithms Using MA Orderings,” Technical Report METR 2004-27, University of Tokyo, 2004.

[54] A. W. McMasters and T. M. Mustin, “Optimal Interdiction of a Supply Network,” Naval Research Logistics Quarterly 17, 261-268, 1970.

[55] R. H. Möhring, A. S. Schulz, F. Stork and M. Uetz, “Solving Project Scheduling Problems by Minimum Cut Computations,” Management Science 49, 330-350, 2003.

[56] H. Nagamochi and T. Ibaraki, “Computing Edge-Connectivity in Multigraphs and Capacitated Graphs,” SIAM Journal of Discrete Mathematics 5, 54-66, 1992.

[57] N. Nagy and S. Akl, “The Maximum Flow Problem: A Real-Time Approach,” Parallel Computing 29, 767-794, 2003.

[58] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization*, John Wiley and Sons Inc., 1988.

[59] F. Ordóñez and J. Zhao, "Robust Capacity Expansion of Network Flows," Networks 50, 136-145, 2007.

[60] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover, 1982.

[61] J. C. Picard and M. Queyranne, "On the Structure of All Minimum Cuts in a Network and Applications," Mathematical Programming Study 13, 8-16, 1980.

[62] D. Pisinger, "A Minimal Algorithm for the 0-1 Knapsack Problem," Operations Research 45, 758-767, 1997.

[63] C. Phillips, "The Network Inhibition Problem," Proceedings of the 25th Annual ACM Symposium on the Theory of Computing (STOC) 776-785, 1993.

[64] J. Puchinger, G. Raidl and U. Pferschy, "The Multidimensional Knapsack Problem: Structure and Algorithms," Technical Report, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2007.

[65] H. D. Ratliff, G. T. Sicilia and S. H. Lubore, "Finding the n Most Vital Links in Flow Networks," Management Science 21, 531-539, 1975.

[66] J. C. Régin, "A Filtering Algorithm for Constraints of Difference in Constraint Satisfaction Problems," In the Proceedings of the Twelfth National Conference on Artificial Intelligence 1, 362-367, 1994.

[67] J. Royset and R. K. Wood, "Solving the Bi-objective Maximum-Flow Network-Interdiction Problem," INFORMS Journal on Computing 19, 175-184, 2007.

[68] T. Santoso, S. Ahmed, M. Goetschalckx and A. Shapiro, "A Stochastic Programming Approach for Supply Chain Network Design Under Uncertainty," European Journal of Operations Research 167, 96-115, 2005.

[69] M. G. Scutellá, "A Note on the Parametric Maximum Flow Problem and Some Related Reoptimization Issues," to appear in Annals of Operations Research, 2005.

[70] M. S. Sodhi, "What about the 'O' in O.R.?" OR/MS Today, p. 12, December 2007.

[71] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Transactions on Software Engineering 3, 85-93, 1997.

[72] D. Strickland, E. Barnes and J. Sokol, "Optimal Protein Structure Alignment Using Maximum Cliques," to appear in Operations Research, 2008.

[73] R. Tarjan, J. Ward, B. Zhang, Y. Zhou and J. Mao, "Balancing Applied to Maximum Network Flow Problems," Lecture Notes in Computer Science 4168, 612-623, 2006.

[74] A. Uygun, "Network Interdiction by Lagrangian Relaxation and Branch-and-Bound," Master's Thesis, Naval Postgraduate School, 2002.

[75] S. Wallace, "Investing in Arcs in a Network to Maximize the Expected Max Flow," Networks 17, 87-103, 1987.

[76] R. Wollmer, "Investments in Stochastic Maximum Flow Networks," Annals of Operations Research 31, 457-467, 1991.

[77] R. K. Wood, "Deterministic Network Interdiction," Mathematical Computational Modeling 17, 1-18, 1993.

[78] J. Xue and Q. Cai, "Profile-Guided Partial Redundancy Elimination Using Control Speculation: A Lifetime Optimal Algorithm and an Experimental Evaluation," Technical Report UNSW-CSE-TR-0420, School of Computer Science and Engineering, University of New South Wales, 2004.

[79] J. Xue and J. Knoop, "A Fresh Look at Partial Redundancy Elimination as a Maximum Flow Problem," In the Proceedings of the International Conference on Compiler Construction, 139-154, 2006.

VITA

Doug Altner received a Bachelor's of Science in mathematics with a concentration in operations research from Carnegie Mellon University in 2003. After graduating with college and university honors, he immediately enrolled at the H. Milton Stewart School of Industrial and Systems Engineering at the Georgia Institute of Technology to pursue a doctorate in industrial and systems engineering. He has worked under the supervision of Dr. Özlem Ergun on a wide variety of problems in network flows and neighborhood search, most of which has culminated into this dissertation.

Doug's research interests are in the algorithmic and computational aspects of combinatorial optimization, network flows, network interdiction and very large-scale neighborhood search. His research has been presented at several different scholarly conferences, including the International Symposium on Mathematical Programming, CPAIOR: The International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combinatorial Optimization Problems, the INFORMS Optimization Society Annual Meeting and the INFORMS Annual Meeting. In August of 2008, He will be joining the Department of Mathematics at the United States Naval Academy.