Rapidly Solving an Online Sequence of Maximum Flow Problems

Doug Altner
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology
765 Ferst Drive NW, Atlanta, Georgia 30332-0205
daltner@isye.gatech.edu

Özlem Ergun
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology
765 Ferst Drive NW, Atlanta, Georgia 30332-0205
oergun@isye.gatech.edu

February 29, 2008

Abstract

We investigate how to rapidly solve an online sequence of maximum flow problems. Sequences of maximum flow problems arise in a diverse collection of settings, including stochastic network programming and real-time scheduling of jobs on a two-processor computer. In this paper, we formulate solving an online sequence of maximum flow problems as the Maximum Flow Reoptimization Problem, introduce a maximum flow algorithm designed for "warm starts," and present computational results.

1 Introduction

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, it seems intuitive that there would be no substantial cost to using a black-box maximum flow solver to solve a sequence of MFPs. The goal of this paper, however, is to convince the reader that this may lead to an enormous number of unnecessary computations.

Sequences of maximum flow problems arise as part of greater computational routines in many different areas of operations research literature.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 [10], 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 [4]. Their algorithm requires ($O(n^4\ (log\ n + n\ log\ U + log\ M)$) maximum flow computations on a series

of bipartite networks. This equilibrium problem has nontrivial applications in transmission control protocol (TCP) congestion control [22].

- *Bicriteria Network Interdiction* In [27], 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 [29], 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 [26], Régin demonstrates how to satisfy the `alldifferent` constraint in Constraint Programming by solving a bipartite matching problem. Thus, to iteratively check if different solutions satisfy a series of `alldifferent` 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.

- *Real-Time Process Scheduling* In [30], 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 [2], Altner shows 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 [21] and two-processor scheduling [30].

- *Separating Valid Inequalities* In [6], 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 requires $O(n^{2p})$ 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 [3] and Carey and Hendrickson [5], 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 [32] and Wallace [31]) as well as a stochastic network interdiction problem where the goal is to minimize the expected maximum flow (e.g., Cormican et al. [9]).

In the aforementioned applications, the maximum flow problems are typically topologically similar. That is, the next maximum flow problem 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 maximum flow problems.

An effective strategy towards quickly solving an entire online sequence of optimization problems is to develop efficient reoptimization heuristics. To this end, we will develop a modified Goldberg-Tarjan algorithm for computing a maximum flow that is designed for efficient "warm starts." We focus on the algorithm of

Goldberg and Tarjan because it is considered the fastest algorithm for computing a maximum flow in practice [8] and because there exists nice properties of the algorithm, which will be discussed in this paper, that are conducive to reoptimization.

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, $|A_{i-1} \oplus A_i| = |A_{i-1} \backslash A_i| + |A_i \backslash A_{i-1}|$. 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 [30]. Third, MFSAROP is a sub-problem encountered when computing a robust minimum cut, with respect to a polyhedral uncertainty set, using the algorithm of Altner in [2].

When approaching a sequence of maximum flow problems, we elected to modify a Goldberg-Tarjan algorithm as opposed to a Edmonds-Karp algorithm. This is primarily because, as discussed in [1] and [8], the Goldberg-Tarjan algorithm is considered the fastest maximum flow algorithm in practice and we have a promising strategy to reoptimize using the push-relabel ideas of this algorithm.

This paper offers many contributions. First, we offer an algorithm to solve MFROP that exploits a cut decomposition. Second, we demonstrate the significant potential savings from using our algorithm as opposed to using a black-box maximum flow solver.

In subsection 1.3, we briefly survey related literature. In section 2, we list pertinent network programming preliminaries. In section 3, we discuss MFSAROP and our algorithmic approach. In section 4, we present our computational results. In section 5, we draw conclusions and discuss future work.

1.3 Related Work

This section surveys work that is similar in nature to maximum flow reoptimization. In [13], 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 [14], [28] and [33].

MFROP is fundamentally different from PFMP. First of all, the sequence of sub-networks 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 [23], 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 [27], 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 [11] 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.

2 Preliminaries

2.1 Notation

We use $N = (V, A)$ to denote a network or directed graph with node set V and arc set A. An arc from node i to node j will be denoted as (i, j). $x_{i,j}$ and $c_{i,j}$ will be used to denote the flow on and the capacity of arc (i, j), respectively. Every network mentioned in this paper 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 paper 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.

Given a node v, the *forward star* of v, that is, the set of all arcs leaving v, will be denoted by $FS(v)$. Similarly, the set of all arcs entering v is known as the *reverse star* and will be denoted by $RS(v)$.

2.2 Network Flow Preliminaries

For background in network flows, we recommend the textbook [1]. 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 [12]. 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.

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 1. *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.*

2.2.1 Goldberg-Tarjan Algorithm

We refer to the well known maximum flow algorithm of Goldberg and Tarjan [17] 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 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 1.

Algorithm 1 Goldberg-Tarjan Algorithm

```
Initialize d(v) and e(v) ∀ v ∈ V
x_e ← c_e ∀ e ∈ FS(s)
x_e ← 0 ∀ e ∉ FS(s)

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

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: $d(i) \leftarrow \min \{d(j) + 1 : (i, j) \in$ residual $FS(i)\}$ denotes relabeling.

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 [8] and their discussion is beyond

the scope of this paper.

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 [8]. We also implemented both the global and the gap relabeling heuristics.

3 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 [24].

3.1 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 $N' = (V, A \backslash \{e'\})$.

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

3.1.1 Hardness Results

Theorem 2. *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 [19].

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 1 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.

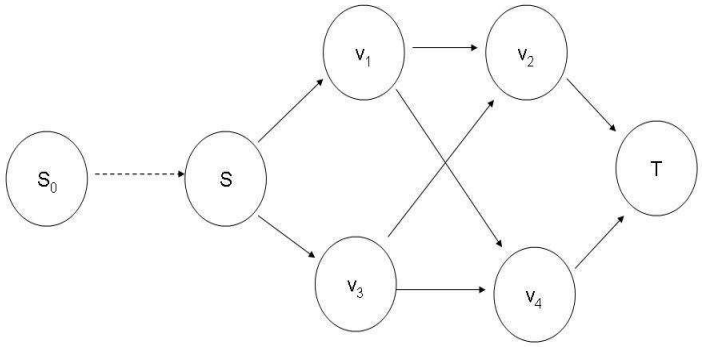

Figure 1: Constructed instance for NAMFRP given a MFP with six nodes.

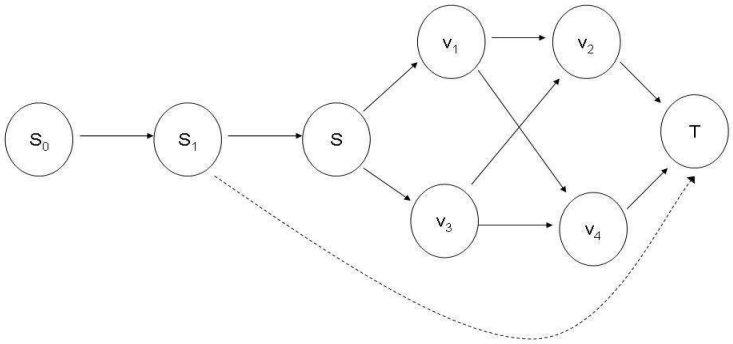

Figure 2: Constructed instance for RAMFRP given a MFP with six nodes.

Theorem 3. *Recomputing the maximum flow after removing a single arc is P-hard.*

Proof: The proof will be a log space reduction from the Maximum Flow Problem, which is shown to be P-hard in [19].

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. □

Please see Figure 2 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.

3.1.2 Algorithmic Results

Theorem 4. *Given Both NAMFRP and RAMFRP can be solved in* $O(c_{e'}m)$.

Proof: First, we will consider NAMFRP, where a new arc e' with capacity $c_{e'}$ is added to a network N. Thus, we may upper-bound the amount that the maximum flow will increase by $c_{e'}$.

An augmenting path in a residual network with m arcs may be found in $O(m)$. 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 $O(c_{e'}m)$.

A similar argument may be made for RAMFRP after all of the s-t paths involving arc e' in the flow-path decomposition of the maximum flow in the previous network are removed with breadth-first search, which is an $O(m)$ operation. The result follows. □

Corollary 5. *MFSAROP may be solved in* $O(G_r + c_{max}mk)$ *where* $G_r = min(n^{\frac{2}{3}}, \sqrt{m})\ m\ log(\frac{n^2}{m})\ log(c_{max})$, $k+1$ *is the total number of maximum flow problems that must be solved and* $c_{max} = max\{c_e | e \in A\}$.

Proof: In [16], Goldberg and Rao prove that a maximum flow may be computed in time $O(G_r)$. This is the worst-case complexity bound on the first maximum flow problem that must be computed, which must be computed from scratch. The theorem above demonstrates that given the maximum flow in a network, the new maximum flow value after a single arc e is added or deleted may be computed in $O(mc_e)$. Since c_{max} is the maximum capacity on any arc in the ground network and there are k subsequent maximum flow reoptimizations that must take place, the result follows. □

3.2 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.

3.2.1 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.

3.2.2 Storing Minimum 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 efficiently store up to two minimum cuts.

Definition 1. *Given a network that is currently at maximum flow, a* cut tripartition *is a tripartition (V_s, V_i, 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 s in the optimal residual network. V_t is the set of all nodes that can currently reach the sink in the optimal residual network. $V_i = V \backslash (V_s \cup V_t)$.*

A cut tripartition implicitly stores two, not necessarily unique, minimum cuts: $(V_s, V \backslash V_s)$ and $(V \backslash V_t, V_t)$. Moreover, this data structure can indicate, in constant time, if a newly added arc is contained in all minimum cuts. Specifically, a newly added arc is contained in all minimum cuts if and only if it originates from a node in V_s and it terminates at a node in V_t.

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.

3.2.3 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 2. 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 2 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 3, 5, 6 and 7 respectively. Given our cut tripartition is already stored, we

Algorithm 2 Maximum Flow Reoptimizer Main

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

$x_1^* \leftarrow$ `GoldbergTarjan(N1)`

`constructCutTripartition(N1,x)`

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

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: $N_i = (V, A_i)$.

The subroutine for the case when a new arc (u, v) is added to all minimum cuts is detailed in Algorithm 3. 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 3 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(Ni, 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(Ni, x, Δub)`. 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.

The pseudocode for `modMaxFlow(N, x, Δub)` is contained in Algorithm 4. `modMaxFlow(N, x, Δub)` 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, Δub)`, 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 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-Tarjab 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

Algorithm 4 `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
$e(\bar{s}) \leftarrow \Delta_{ub}$ $x_{(\bar{s},s)} \leftarrow \Delta_{ub}$

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)`

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 3 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 5 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 5. 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.

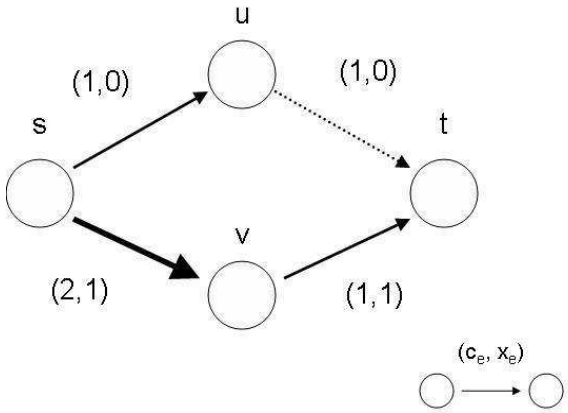

Figure 3: Saturating the "wrong" arc.

Algorithm 6 Case III: Deleting an arc (u, v) that is not in a stored min. cut

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

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

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

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

`expand(`$\mathtt{N}_i$`,` $\mathtt{t}_v$`)`

if $e(v) < 0$ **then**

`removeGhostFlow(`$\mathtt{N}_i$`, v, e(v))` // Remove any remaining ghost flow from the network.

end if

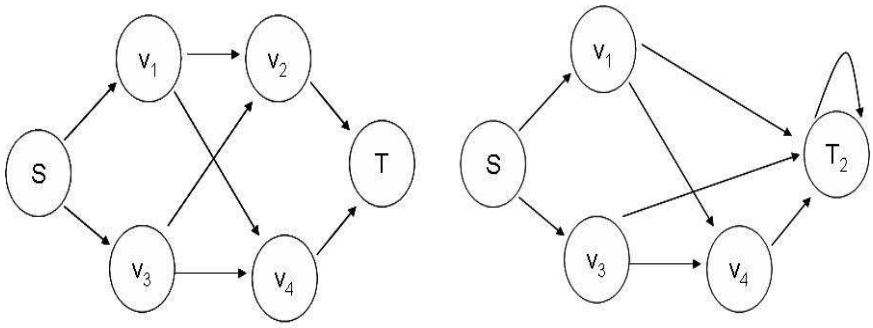

Figure 4: Identifying v_2 and t to form a new sink t_2; before and after.

The subroutine for the case when an arc (u, v) is deleted that is not in a stored minimum cut is detailed in Algorithm 6. 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:

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 are often called *loops* in graph theory literature. Please see Figure 4 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 8.

The subroutine for the case when a deleted arc (u, v) is in at least one minimum cut 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 at least one of the following two booleans is true:

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

Algorithm 7 Case IV: Deleting an arc (u, v) that is in a stored min cut

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

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

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

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

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

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.

Algorithm 8 `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 $\leftarrow$ 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

3.3 Extensions of Single Arc Reoptimization

This section details extensions of our algorithm for MFSAROP.

3.3.1 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 $c_{\bar{e}} = d_e$.

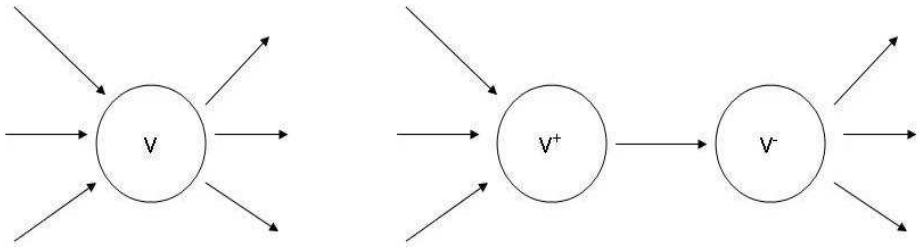

Figure 5: Splitting a Node; Before and After.

The modified network may then be reoptimized using Algorithm 2. After reoptimization is complete, the two parallel arcs are *merged* into a single arc.

Definition 2. *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 $x_{e_2} = max\{0, x_e + d_e - c_e\}$. One may then remove arc e_2 and reoptimize accordingly.

3.3.2 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.

3.4 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 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. 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 [8]. 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.

Lastly, we chose not to include any results using a major commercial linear programming package such as CPLEX. When used as a black-box, Goldberg's implementation of the Goldberg-Tarjan algorithm [15] is typically faster than warm starting commercial software, including CPLEX 9.0, for solving a sequence of maximum flow problems. This being considered, we restricted our computational study to testing network-based algorithms for the maximum flow problem.

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.

4.1 Generic Maximum Flow 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 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

Table 1: 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%

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`.

Table 1 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

Table 2: 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%

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.

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 2 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.

5 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 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 [25] 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 2 for MFSAROP. With such a structure, we can use Algorithm 7 for many instances of delete arc reoptimization instead of the slower subroutine Algorithm 6.

Given the problem statement of the Maximum Flow Reoptimization Problem, it is natural to consider a network simplex approach. We suspect that this may not be desirable for several reasons. First of all, the network simplex algorithm is generally accepted as being much slower than the Goldberg-Tarjan algorithm. Through some initial computational tests, the authors discovered that in a restricted MFROP problem where only arcs are added, our reoptimization code overwhelmingly outperformed a network simplex algorithm. We conjecture that degenerate simplex pivots greatly contributed to the sluggish performance of the network simplex code.

Unless dual simplex pivots can greatly outperform our reoptimization code after arcs have been removed, we see no reason to believe that a modified network simplex method would be more desirable for MFROP. Nevertheless, we encourage future research in implementing such a simplex algorithm, possibly based on those in [18] and [20].

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 paper will help advance understanding and spark interest in this area of research.

Acknowledgements: The authors would like to thank James Orlin for a helpful observation.

Özlem Ergun was partially supported by the NSF Career grant DMI-0238815.

References

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

[2] D. Altner, "An Efficient Algorithm for Computing Robust Minimum Capacity s-t Cuts," in preparation, 2008.

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

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

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

[6] R. Carr, "Separating Clique Trees and Bipartition Inequalities Having a Fixed Number of Handles and Teeth in Polynomial Time," Mathematics of Operations Research, 22(2):257-265, 1997.

[7] 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.

[8] B. Cherkassky and A. Goldberg, "On Implementing Push-Relabel Method for the Maximum Flow Problem," Algorithmica, 19(4): 390-410, 1994.

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

[10] N. Devanur, C. Papadimitriou, 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, 2002.

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

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

[13] A. Frangioni and A. Manca, "A Computational Study of Cost Reoptimization for Min-Cost Flow Problems," INFORMS Journal on Computing, 18(1):61-70, Winter 2006.

[14] G. Gallo, M. Grigoriadis and R. Tarjan, "A Fast Parametric Maximum Flow Algorithm and Applications," SIAM Journal of Computing, 18(1):30-55, 1989.

[15] A. Goldberg, "Andrew Goldberg's Network Optimization Library," `http://avglab.com/andrew/soft.html`.

[16] A. Goldberg and S. Rao, "Beyond the Flow Decomposition Barrier," Journal of Associated Computing Machinery, 45, 783-797, 1998.

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

[18] D. Goldfarb and W. Chen, "On Strongly Polynomial Dual Simplex Algorithms for the Maximum Flow Problem," Mathematical Programming, 78:159-168, 1997.

[19] 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.

[20] D. Hochbaum, "The Pseudoflow Algorithm and the Pseudoflow-Based Simplex for the Maximum Flow Problem," Lecture Notes in Computer Science, Proceedings of the 6th International IPCO Conference, 325-337, 1998.

[21] D. Hochbaum and A. Chen, "Improved Planning for the Open - Pit Mining Problem," Operations Research, 48, 894-914, 2000.

[22] F. P. Kelly, "Charging and Rate Control for Elastic Traffic," European Transactions on Telecommunications, 8:33-37, 1997.

[23] N. Nagy and S. Akl, "The Maximum Flow Problem: A Real-Time Approach," Parallel Computing, 29(6):767-794, 2003.

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

[25] 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.

[26] 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.

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

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

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

[30] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Transactions on Software Engineering, 3(1):85-93, 1977.

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

[32] R. Wollmer, "Investments in Stochastic Maximum Flow Networks," Annals of Operations Research, 31(1):457-467, 1991.

[33] B. Zhang, J. Ward and Q. Feng, "A Simultaneous Parametric Maximum-Flow Algorithm for Finding the Complete Chain of Solutions," Technical Report, Hewlett-Packard, 2004.