Implementations of the Pseudoflow Algorithm for the Maximum Flow Problem

Charles L. Anderson

February 2007

Forward

This document is a work-in-progress version of my PhD dissertation that I last worked on in 2007, shortly before my mother died and my first child was born. This document went through years of revisions without converging to solution. I opted to walk away from it rather than continuing to spin my wheels.

Contents

List of Figures

List of Tables

Abstract

The maximum flow problem is one of the most fundamental problems in network optimization. To date, the most efficient algorithm in theory and practice has been the push-relabel algorithm of Goldberg and Tarjan. Recently, Hochbaum introduced an algorithm with comparable complexity—the pseudoflow algorithm. Prior to this work, its practical performance had been unknown.

Both the pseudoflow and push-relabel algorithms can efficiently solve a specific type of parametric maximum flow problem described by Gallo, Grigoiadis, and Tarjan where the capacities of the source and sink arcs vary monotonically based on a single parameter value. Both algorithms can solve a parametric instance with a series of parameter values with the same worst-case complexity as a single run. The empirical performance of these algorithms has not previously been studied.

A unique feature of the pseudoflow algorithm is that it can also work with more general variations of the arc capacities. The capacities of any arc in the graph can vary, independent of a parameter. The pseudoflow algorithm supports a warm start technique whereby the solution from one problem instance is used as a starting point for the solution to the next instance.

This work provides a detailed study of the performance of implementations of the pseudoflow, push-relabel, and Dinic algorithms using a wide range of problem instances, including both common synthetic instances and application data. First, we present performance comparisons with single instances of maximum flow problems. We also discuss the effects of the large number of heuristics available on the pseudoflow solver. Next, we study the practical performance of the first implementations of the parametric pseudoflow and push-relabel solvers. Finally, we study the performance of the warm start pseudoflow solver.

Chapter 1

Introduction

The maximum flow problem is one of the most fundamental problems in network optimization that arises in a wide variety of applications. Due to its applicability, numerous algorithms have been developed to solve it. These algorithms can be divided into two major classes: feasible flow algorithms and preflow algorithms. Feasible flow algorithms increase the flow at each step along augmenting paths, and they maintain a feasible flow at each step. Preflow algorithms permit the inflow of a node to exceed the outflow during the intermediate steps of the algorithm. The pseudoflow algorithm of Hochbaum [Hoc97] differs distinctly from feasible flow and preflow algorithms in that it works with pseudoflows. The pseudoflow algorithm removes constraints on the inflow and outflow of a node so that the inflow may exceed the outflow, or the outflow may exceed the inflow. This flexibility leads to many heuristics that allow the implementation to be tuned to various types of problems.

Prior to the pseudoflow algorithm, the most efficient algorithm in theory and practice has been the push-relabel algorithm [CG95b], which is a preflow algorithm. Numerous authors implemented the push-relabel algorithm, and various heuristics and techniques were developed to maximize its performance. Before this work, the practical performance of the pseudoflow algorithm has been untested. We compare the performance of the pseudoflow algorithm to the push-relabel algorithm (the best preflow algorithm) and Dinic's algorithm (the best feasible flow algorithm). We evaluate its practicality on a set of common benchmark problem instances (the DIMACS problems), as well as some application instances. We also investigate many heuristics made possible by the flexibility of the pseudoflow

algorithm.

In a specific parametric form of the maximum flow problem, the capacities of the arcs out of the source and into the sink are monotone functions of a single parameter. Gallo, Grigoiadis, and Tarjan [GGT89] showed how the push-relabel algorithm could be used to solve parametric maximum flow problems efficiently with the same complexity as a single run of the push-relabel algorithm. Hochbaum [Hoc97] showed how the pseudoflow algorithm can also be used to solve the parametric maximum flow problem with the complexity of a single run. We test implementations of the parametric algorithms for both push-relabel and pseudoflow.

A unique feature of the pseudoflow algorithm is that it can use any pseudoflow to initialize the algorithm. In particular, a previous solution to a related problem may be used. We developed a warm start technique that uses the solution from one instance as a starting point for the next instance. This allows the capacities of the arcs in a series of related problems to vary arbitrarily—i.e., the variations are not restricted to simple parametric variations.

In this dissertation, we present the maximum flow problem and outline some algorithms for solving it. We present the pseudoflow algorithm in some detail. The bulk of this work focuses on the performance of implementations of the pseudoflow algorithm compared to the push-relabel algorithm. We also compare the performance of the pseudoflow and push-relabel algorithms for solving the parametric maximum flow problem. Finally, we examine the viability of the warm start technique for the pseudoflow algorithm. A detailed description of the contributions of this dissertation can be found in Section 1.5.

1.1 Definitions

Let $G = (V, A)$ be a directed network with nodes V and arcs A. We denote the number of nodes as n and the number of arcs as m. There are two distinguished nodes in the network: the source node s and the sink node t.

For a node i, we call the set of all nodes j such that $(i, j) \in A$ or $(j, i) \in A$ the *neighbors* of i. An arc (u, v) of unspecified direction is referred to as *edge* $[u, v]$. We say that $[u, v] \in A$ if $(u, v) \in A$ or $(v, u) \in A$.

A *directed path* from v_1 to v_k, denoted $(v_1, v_2, \ldots, v_k)$, is an ordered sequence of distinct nodes such that $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in A$. A *u-v path* is a directed path from u to v. $[v_1, v_2, \ldots, v_k]$ denotes an *undirected path* from v_1 to v_k where $[v_1, v_2], \ldots, [v_{k-1}, v_k] \in A$.

1.1.1 Maximum Flow Problem

For each arc (u, v) in the graph, we associate a finite, non-negative *capacity*, $c(u, v)$. Let us assume that every node v is on some directed path from s to t. This implies that $m \geq n - 1$. Let us also assume that if $(u, v) \in A$, then $(v, u) \notin A$.

We can state the generic maximum flow problem as follows:

$$\begin{aligned} &\text{Maximize} \quad v \\ &\text{subject to:} \quad \sum_{j:(i,j)\in A} f(i,j) - \sum_{j:(j,i)\in A} f(j,i) = \begin{cases} v & \text{if } i = s \\ -v & \text{if } i = t \\ 0 & \text{if } i \in V \setminus \{s,t\} \end{cases} \\ &\qquad\qquad 0 \leq f(i,j) \leq c(i,j) \qquad\qquad \forall (i,j) \in A \end{aligned}$$

It is well known (e.g., [AMO93]) that without loss of generality, the maximum flow problem can be generalized and reformulated to include variations such as non-zero lower bounds on the arcs.

A vector f of values $f(i, j)$, $(i, j) \in A$, that meets the constraints is called a *feasible flow*. The first set of equality constraints is called the *flow conservation* constraints. The second set of inequality constraints is called the *capacity* constraints. The scalar v is called the *value* of the flow.

A *preflow* is a vector f that meets the capacity constraints with a weakened form of the flow conservation constraints: $\sum_{j:(j,i)\in A} f(j, i) - \sum_{j:(i,j)\in A} f(i, j) \geq 0$ for all nodes i except the source and sink. A *pseudoflow* is a vector f that only satisfies the capacity constraints.

The *excess* of a node i with respect to a given vector f is defined as $e_f(i) = \sum_{j:(j,i)\in A} f(j, i) - \sum_{j:(i,j)\in A} f(i, j)$. A excess with a negative value is referred to as a *deficit*.

1.1.2 Residual Graph

Let us define a *residual graph* with respect to a graph G and a pseudoflow f as follows. We replace each arc (u, v) in the original network with two arcs (u, v) and (v, u). The

arcs each have a *residual capacity*: $r_f(u,v) = c(u,v) - f(u,v)$ and $r_f(v,u) = f(u,v)$. The *residual network* with respect to a network G and a pseudoflow f (written G_f) is the network with capacities r_f that contains only arcs with positive residual capacity. An *augmenting path* in G_f is a simple, directed s-t path in G_f.

1.1.3 Minimum Cut

Given a partition of the nodes in V into two sets S and T, a *cut* is the set of arcs with one endpoint in S and the other in T, and we write this as (S,T). An *s-t cut* is a cut such that $s \in S$, and $t \in T$. S is referred to as the *source set* of the s-t cut, and T is referred to as the *sink set* of the s-t cut.

Given a flow f, the *value of a flow across a cut* is the net flow of the arcs from nodes in S to nodes in T—i.e., $f(S,T) = \sum_{(u,v)\in(S,T)} f(u,v) - \sum_{(v,u)\in(T,S)} f(v,u)$. Similarly, the *capacity of a cut* is $c(S,T) = \sum_{(u,v)\in(S,T)} c(u,v)$.

A *minimum s-t cut* is an s-t cut with minimum capacity. Each graph contains at least one such cut, but there may be more than one cut with minimum value.

The connection between the maximum flow and the minimum cut was established by Ford and Fulkerson [FF56]. The maximum-flow minimum cut theorem says that the maximum possible value of an s-t flow is equal to the minimum capacity of all possible s-t cuts. It is well known (e.g., [AMO93]) that given a maximum flow it is easy to obtain a minimum cut. So, the minimum cut problem can be solved by first solving the maximum flow problem—see Section 1.2 for examples.

1.2 Applications

Part of the reason that the maximum flow problem has been such a popular area of research is that there is a large number of applications that can be formulated and solved via maximum flow or minimum cut. Below is a sample list of applications:

- Danzig and Fulkerson [DF54] modeled tanker scheduling as a maximum flow problem, where the object is to ship perishable goods between several origin-destination pairs using the fewest number of ships subject to delivery dates.

- Levien and Aiken [LA98] showed how to model and analyze the trust metric of a certificate chain used in public-key certification as a maximum flow problem.

- Stone [Sto78] discussed the problem of assigning program modules or subroutines to processors in a two-processor computer system. The goal is to minimize the communication between processors.

- Johnson and Walz [JW86] applied a maximum flow technique to the set of type constraints in programming language compilers in order to identify the most likely source of type errors. This can be used to generate error messages or to recommend alternative types.

- Chase and Garg [CG95a] discussed detecting predicates or conditions in distributed systems. The general problem is NP-complete, but they showed how bounded sum predicates can be detected using a maximum flow formulation.

- Johnson [Joh68] formulated the open pit mining problem as a minimum cut problem, where the objective is to determine the optimal contour of an open pit mine, subject to precedence constraints on the blocks that must be removed.

- Federgruen and Groenevelt [FG86] modeled the problem of scheduling jobs on uniform parallel machines as a maximum flow problem. In this problem there is a set of jobs and a set of uniform machines to carry out the jobs. Each job has a processing time requirement, and jobs can be preempted. The objective is to find a feasible schedule for the jobs or show that none exists.

- Another scheduling problem was described by Möhring et al. [MSSU03]. The problem consists of a series of jobs to be scheduled over a series of time periods, and there are precedence relationships between the jobs. The authors used minimum cut as a subroutine in a Lagrangian relaxation solution technique. (See Section 3.4.2.7 for details.)

- Wong and Yang [YW94] presented a heuristic to solve the two-way, balanced partitioning (bipartitioning) problem, which is NP-complete. The problem arises in

VLSI design, where the objective is to partition a circuit into two balanced components while minimizing the number of crossing nets (sets of edges in the cut). Their heuristic solves a sequence of minimum s-t cut problems.

- Goldschmidt and Hochbaum [GH94] presented another partitioning problem where the objective is to partition an edge-weighted graph into k non-empty components such that the total edge weights between components is minimum. This problem is NP-complete for arbitrary values of k. For a fixed value of k, they presented a polynomial time algorithm that enumerates minimum cuts.

- Eisner and Severence [ES76] developed a model for segmenting a large, shared database between primary and secondary memory. This is solved with a series of minimum cut problems.

- Hochbaum and Pathria [HP97] described two forest-harvesting problems where the forest area is partitioned into cells. Benefit is derived from harvesting a cell, and either a penalty is paid for harvesting an adjacent cell or a benefit is derived from creating a border between harvested cells and unharvested cells.

Interested readers should see Ahuja, Magnanti, and Orlin [AMO93] for a more extensive list of applications.

1.3 A Brief History of Algorithms for Maximum Flow

In this section, we look at the progression of algorithms over the last five decades to solve the maximum flow problem. The list is far from complete. Rather, it is a survey of significant algorithms that introduced new ways of solving the problem and that typically ran faster than previous algorithms.

For each algorithm, we provide a brief outline of the algorithm and its complexity. For more detailed presentations, the reader should consult the references.

1.3.1 Ford and Fulkerson—1956

Ford and Fulkerson's algorithm [FF56] is the first algorithm for solving the maximum flow problem. It is based on the idea of sending flow along augmenting paths in the

residual graph. Once an s-t path $P = (s, v_1, v_2, \ldots, v_k, t) \in G_f$ is identified, as much flow as possible is pushed along the path. The amount of the flow is the minimum residual capacity of the arcs in the path—i.e., $\delta = \min\{r_f(u, v) : (u, v) \in P\}$. An arc (u, v) with residual capacity δ is known as a *bottleneck arc* (note that there may be more than one arc with residual capacity δ). After pushing δ units along P, all of the bottleneck arcs will be saturated—i.e., they will have residual capacity of zero in that direction.

To identify the augmenting path, we can use any search technique from the source that visits each node at most once (e.g., depth-first or breadth-first search). This requires $O(m)$ operations because we scan each residual arc at most once. Augmenting along the path requires $O(n)$ operations. If we assume that the arc capacities are integral, and the maximum capacity of any arc is U, then the maximum value of the flow is nU. Each augmentation sends at least one unit of flow. Therefore, the worst case complexity is $O(nmU)$, which is pseudo-polynomial because it depends on U.

1.3.2 Edmonds and Karp—1972

Edmonds and Karp [EK72] improved on Ford-Fulkerson by choosing the shortest augmenting path (expressed in number of arcs) along which to send flow. Before each augmentation, the algorithm performs a breadth-first search in the residual graph from the source to identify the shortest path from the source to the sink.

Each augmentation saturates at least one arc. It can be shown that during the execution of the algorithm, an arc (u, v) can be saturated at most $O(n)$ times because between two successive saturations of (u, v), the distance from s to t in the residual graph must increase by at least two units, and the maximum distance is n. Since each augmentation saturates at least one arc, there can be at most $O(mn)$ augmentations.[1] Each iteration requires $O(m)$ operations to perform the breadth-first search to identify the shortest augmenting path and $O(n)$ operations to augment along the shortest path. The complexity of each iteration is $O(m)$. Therefore, the overall complexity is $O(nm^2)$, which is no longer dependent on the maximum capacity of an arc in the graph. This was the first strongly polynomial algorithm for the maximum flow problem.

[1] Edmonds and Karp state this result as $O(n^3)$ augmentations because they assume the number of arcs is $O(n^2)$—i.e., they do not use a separate variable for the number of arcs.

1.3.3 Dinic—1970

Dinic's algorithm [Din70] augments along shortest paths, but it does so without recomputing the shortest path distances after each augmentation. Rather, it augments along all paths of a specific length. It does this by constructing a series of *layered networks*. A layered network is constructed from the residual graph by labeling each node v with the exact shortest path distance from v to t in G_f. This distance is denoted $D_f(v)$. Then, the network is pruned to remove all arcs (u, v) that do not meet the condition $D_f(u) = D_f(v)+1$. In this network, all paths from s to t are shortest paths with the same length, $D_f(s)$. This is called a layered network because if the nodes are partitioned into sets L_i such that all nodes in L_i have labels $D_f(v) = i$, then all arcs in the network (u, v) are such that $u \in L_i$ and $v \in L_{i-1}$. The sets of nodes L_i are called *layers*. Note that this network is clearly acyclic.

A flow in a layered network such that there is no augmenting path from s to t is called a *blocking flow*. A blocking flow is a maximum flow in a given layered network, but it is not necessarily a maximum flow in the original network.

The algorithm augments flow along s-t paths in the layered network until no augmenting path remains. The flow at this stage is a blocking flow. Although the algorithm updates the residual capacities of the arcs in the layered network after each augmentation, it does not immediately update the residual capacities of the reverse arcs in the residual graph (which are not present in the layered network) nor does it update the shortest path distances of the nodes. Each augmentation requires time $O(n)$ because no path is longer than n. There are at most $O(m)$ augmentations in a given layered network because each augmentation saturates at least one arc. Then another layered network is constructed from the new residual graph, and the process is repeated. There can be at most $O(n)$ layered networks because the longest path from s to t contains $n - 1$ arcs. Therefore, the overall complexity is $O(n^2m)$. Malhotra, Kumar, and Maheshwari [MKM78] developed a variant of Dinic's algorithm that has complexity $O(n^3)$.

1.3.4 Karzanov—1974

Karzanov [Kar74] also used the concept of layered networks; however, the Karzanov algorithm was the first that was not an augmenting path algorithm.[2] Instead, it uses preflows.

The Karzanov algorithm is divided into phases, each of which establishes a blocking flow in a layered network. During each phase, the algorithm works with preflows. At the end of a phase and at the end of the entire algorithm, the flow is feasible.

During a phase, the algorithm processes all nodes in a layer with two procedures: *advance* and *balance*. *Advance* pushes excess from all the nodes in a layer to the next layer (closer to the sink), and continues pushing the excess through the layers until the excess reaches the sink or there is no outlet for it.

Balance returns excess from all nodes in a single layer to the previous layer by canceling a subset of the flows into the nodes with positive excess. It only pushes the excess back one layer, which differs from *advance*, which pushes through as many layers as possible. Once a node is balanced, the node is marked as *blocked* so that *advance* will not push any more excess into the node during the current phase. Therefore, *balance* can be called at most $n - 2$ times in a phase.

Each phase consists of alternating applications of *advance* and *balance*, so both operations are performed at most $n - 2$ times. During each application of *advance*, there are at most $O(n)$ non-saturating pushes, because there is only one partially saturated arc per node. Therefore, the total number of non-saturating pushes during the phase is $O(n^2)$. The algorithm establishes a maximum flow in a layered network in $O(n^2)$ operations. There are $O(n)$ layered networks, so the overall complexity is $O(n^3)$.

1.3.5 Goldberg (push-relabel)—1985

Goldberg [Gol85] presented another preflow algorithm, but it does not use layered networks. Instead, flow is pushed from nodes with excess individually, rather than treating them in groups (e.g., layers). With each node v we associate a node label, which is a non-negative integer denoted $d(v)$. The labels are *valid* if $d(s) = n$, $d(t) = 0$, and $d(v) \leq d(w) + 1$ for every residual arc (v, w). If $d(v) < n$, then the label represents a lower

[2] The source for Karzanov's paper [Kar74] is translated from Russian. A good presentation of the algorithm, written in English, can be found in Even [Eve78].

bound on the distance from v to t in the residual graph, and if $d(v) \geq n$, then $d(v) - n$ is a lower bound on the distance from v back to the source in G_f [GT88].

The algorithm begins by saturating all arcs out of the source. Any node with positive excess is called an *active* node, so all nodes adjacent to the source are initially active. The algorithm applies two operations on active nodes: *push* and *relabel*. *Push* sends excess out of a node to its neighbors along residual arcs such that the neighbor has a label one less than the node. If excess cannot be pushed out of a node along such a residual arc, *relabel* increases the label of the node to another valid value.

The algorithm continues applying *push* and *relabel* until it establishes a maximum flow. This basic framework leads to a family of related algorithms known by a number of names including push-relabel, preflow-push, Goldberg's, and Goldberg-Tarjan. The generic algorithm does not specify any order on the push and relabel operations and can be shown (Goldberg [Gol85]) to run in time $O(n^2m)$.

Goldberg and Tarjan [GT88] presented variants with improved complexity. Using a first-in, first-out (FIFO) queue for the active nodes, the algorithm runs in time $O(n^3)$, matching Karzanov's algorithm [Kar74]. With the use of dynamic trees [ST83], the running time of the FIFO algorithm is improved to $O(mn \log(n^2/m))$.

Goldberg and Tarjan recommended additional node selection heuristics without suggesting that they would improve on the complexity of the generic algorithm. These include selecting the active node with the largest excess and using a last-in, first-out (LIFO) stack for the active nodes. Goldberg and Tarjan also suggested selecting the active node with the highest label. This was later shown by Cheriyan and Maheshwari [CM89] to have complexity $O(n^2\sqrt{m})$.

There are two heuristics which are common in implementations of the push-relabel algorithm. The *global relabeling heuristic* was suggested by Goldberg [Gol85, GT88] to improve performance by periodically updating the labels of nodes in the graph based on their distance to the sink in the residual graph. The second, the *gap relabeling* heuristic, was discovered independently by Derigs and Meier [DM89] and by Cherkassky [Che79], as well as Ahuja and Orlin [AO91]. The heuristic identifies nodes known to be in the source set of the minimum cut and raises their labels to n.

1.3.6 Hochbaum—1997

Hochbaum [Hoc97] presented the pseudoflow algorithm for maximum flow motivated by an algorithm of Lerchs and Grossmann [LG65] that solves the maximum closure problem. The algorithm works with pseudoflows rather than flows or preflows. We present a generic algorithm and a number of variants in the next chapter. The complexity of the best variants of the pseudoflow algorithm is $O(mn \log n)$.

1.3.7 Summary

The complexity of the algorithms described is summarized below:

Algorithm	Complexity
Ford and Fulkerson [FF56]	$O(nmU)$
Edmonds and Karp [EK72]	$O(nm^2)$
Dinic [Din70]	$O(n^2m)$
Karzanov [Kar74]	$O(n^3)$
Goldberg and Tarjan [GT88]	$O(mn \log(n^2/m))$
Hochbaum [Hoc97]	$O(mn \log n)$

1.4 Recent Maximum Flow Implementations

In this dissertation, we are primarily concerned with the performance of the pseudoflow algorithm. Recent papers describing previous maximum flow implementations established push-relabel as the fastest algorithm in practice. Therefore, we will focus on comparing the pseudoflow algorithm to the push-relabel algorithm. The results of some of these implementation papers are summarized below—more details can be found in Section 3.1.

The paper of Derigs and Meier [DM89] was one of the earliest to compare the performance of push-relabel to Dinic's algorithm, which had previously been regarded as the best in practice. The authors concluded that push-relabel is superior to Dinic's algorithm. The highest label variant of push-relabel was generally the best.

Anderson and Setubal [AS93] compared the push-relabel algorithm to Dinic's algorithm and implemented older algorithms including Ford-Fulkerson, Karp and Edmonds, and Karzanov. They confirmed that push-relabel was superior to all other algorithms.

Badics and Boros [BB93] made aggressive use of theoretical techniques and data structures (e.g., dynamic trees) and compared the results to simpler implementations. They

found the simple FIFO implementation to be superior in most cases.

Goldberg and Cherkassky [CG95b] implemented Dinic's algorithm and several variants of push-relabel. Their implementations of both algorithms were faster than any previous implementations, so we used their implementations as the baseline for our experiments in Chapter 3.

Ahuja, Kodialam, Mishra, and Orlin [AKMO97] studied a number of algorithms including variants of their own capacity scaling algorithm. They confirmed the results of other authors: the highest label push-relabel algorithm is generally the fastest in practice.

1.5 Contributions

The contributions of this work include:

1. Development of the highest-label pseudoflow algorithm, which is faster in practice than the lowest-label pseudoflow algorithm presented by Hochbaum [Hoc97].

2. Experimental results of the first implementations of the pseudoflow family of algorithms for the maximum flow problem: lowest-label, highest-label, and simplex pseudoflow algorithms.

3. Investigation of heuristics for the pseudoflow algorithm. Some of these are described by Hochbaum [Hoc97]. Others, such as using exact distance labels and the early termination rule, are new in this work.

4. Experimental results of the first implementations of the parametric pseudoflow algorithm and parametric push-relabel algorithms.

5. A warm start pseudoflow algorithm for solving a sequence of maximum flow problems with the same nodes and arcs, where the capacities vary arbitrarily and are not functions of a parameter. We also present experimental results of our implementation.

1.6 Organization of the Dissertation

In the next chapter, we present the pseudoflow family of algorithms. We describe three different algorithms and a number of heuristics that we explored during our implementation.

Chapter 3 contains the result of our experimental studies. We compare the performance of our implementation of the pseudoflow algorithms to the best known implementations of the push-relabel algorithm and Dinic's algorithm.

In Chapter 4, we consider a type of parametric analysis for maximum flow graphs that was first proposed for the push-relabel algorithm by Gallo, Grigoiadis, and Tarjan [GGT89] and for the pseudoflow algorithm by Hochbaum [Hoc97]. We compare the performance of the parametric solvers for both push-relabel and pseudoflow.

Finally, in Chapter 5 we present a technique that is unique to the pseudoflow algorithm, where we solve a series of network instances with the same nodes and arcs, but the capacities change arbitrarily. We use a warm start technique where the solution of one instance is used as a starting point for the next instance.

Chapter 2

Pseudoflow Algorithm

Hochbaum [Hoc97] presented a new algorithm for the maximum flow problem which was motivated by an algorithm from the mining industry by Lerchs and Grossmann [LG65] that solves the maximum closure problem. The maximum flow algorithm works with pseudoflows in which nodes are allowed to have deficits as well as excesses—i.e., the capacity constraints on the arcs are maintained, but the excess of nodes may be negative, zero, or positive.

In this chapter, we provide a description of the generic pseudoflow algorithm as well as the lowest and highest label variants of the pseudoflow algorithm on which our implementations are based. We also describe the heuristics developed for the pseudoflow family of algorithms.

2.1 Definitions

2.1.1 Trees

A *tree* is a connected, acyclic, undirected graph. A *forest* is an acyclic, undirected graph. A *rooted tree* is a tree with a designated *root* node. The edges of a rooted tree define a *parent-child* relationship. Each node, except the root, has a unique *parent*, which is the next node along the unique path in the tree from that node to the root. If v is the parent of u, we say u is a *child* of v. A node with no children is called a *leaf*.

We denote the parent of a node v as $par(v)$. The parent of a root is denoted as *nil*.

A parent-child relationship on a collection of nodes V defines a unique forest of rooted trees. The set of edges of the forest is $\{[u, v] : par(u) = v\}$. The set of root nodes is

$\{v : par(v) = nil\}$.

For convenience in discussions, we adopt the convention that parents are "above" children, the root is at the "top" of a tree, and leaves are at the "bottom." Given a rooted tree with a specified root node, for a node v in the tree, we denote the root of the tree as r_v. An arc along the path from a root to another node in the tree is said to point *downwards*.

In a rooted tree, let the *level* of a node be one plus the number of edges along the path from the node to the root. The level of a root is one, the levels of its children are two, etc.

2.1.2 Normalized Forests

Given a network $G = (V, A)$ with source s and sink t, let $V_I = V \setminus \{s, t\}$; $|V_I| = n - 2$. We call these the *interior nodes* of the graph. Let $G_I = (V_I, E_I)$ be the undirected version of the subgraph induced by V_I. Specifically, $E_I = \{[u, v] \in V_I \times V_I : (u, v) \in A\}$.

Given a parent-child relationship par(v) defined on the nodes of V_I, let $T = (V_I, E_T)$ be a collection of rooted trees in G_I induced by $E_T \subseteq E_I$, where E_T are edges defining the parent-child relationship: $E_T = \{[u, v] : \text{par}(u) = v\}$. We call the rooted trees within T *branches*. An edge $[u, v] \in E_T$ is an *in-tree edge*, and an edge $[u, v] \in E_I \setminus E_T$ is called an *out-of-tree edge*.

Given a pseudoflow f, we define the residual capacity (r_f) with respect to f in G_I and T just as we did for G in Section 1.1.2. The excess $e_f(v)$ of a node v with respect to f is also defined as it was in G.

A forest T of rooted trees in G_I is called a *normalized forest* with respect to a pseudoflow f if it has the following properties:

1. The flows on arcs out of the source and into the sink in G are at their upper capacity bound: $f(s, v) = c(s, v), \forall\ v \in V$ such that $(s, v) \in A$, and $f(v, t) = c(v, t), \forall\ v \in V$ such that $(s, v) \in A$.

2. The flows on all out-of-tree arcs in G_I are at their upper bound or zero.

3. In every branch, all downwards residual capacities are strictly positive.

4. The only nodes in T with nonzero excess are the roots of branches. All other nodes satisfy their flow balance constraints and have zero excess.

Given a branch with a specified root node, we call the branch and all nodes in it *strong* if the root node of the branch has positive excess; we call the branch and all nodes in it *weak* if the root node has non-positive excess.

2.2 Generic Pseudoflow Algorithm

In this section we present the generic pseudoflow algorithm. The algorithm can be divided into three stages: initialization, Phase I, and Phase II. A simple example of the execution of the algorithm can be found in Appendix A.

In later sections, we will present additional pseudoflow algorithms. These algorithms will be variations of Phase I—i.e., the initialization and Phase II stages will be the same for all algorithms.

2.2.1 Initialization

The pseudoflow algorithm allows for many different initialization procedures, so long as the normalized forest properties are satisfied. In this section, we discuss the *simple initialization* scheme. More involved initialization schemes are discussed in Section 2.6.1.

All initialization procedures take the graph (V and A) and the capacities (c) as inputs. The procedures establish an initial pseudoflow (f) and a forest structure defined by a parent-child relationship ($\text{par}(v)$) that conforms to the properties of the normalized forest structure.[1]

The simple initialization procedure saturates the arcs out of the source and those into the sink, as required by the normalized forest properties. All other arcs have zero flow.

With simple initialization, the nodes adjacent to the source have positive excess equal to the capacity of the arcs from the source to the nodes. Similarly, nodes adjacent to the sink have a deficit equal to the capacity of the arc from the nodes to the sink. For a node adjacent to both the source and the sink (i.e., there is an arc from the source to the node and another arc from the node to the sink), we saturate both arcs and sum the excess and

[1] These variable/parameter names are used for all procedures in this chapter.

deficit: $e_f(v) = c(s, v) - c(v, t)$. Such a node could either have net positive, negative, or zero excess.

Below is the pseudocode for the *simpleInit* procedure. The *simpleInit* procedure calls three common subroutines: one to initialize the parent-child relationship to make all nodes root nodes, one to saturate the source and sink arcs—setting the flows on other arcs to zero, and one to mark the branches as strong or weak. These are broken out as separate procedures for use by other initialization routines later in the chapter.

```
procedure simpleInit(V, A, s, t, c):
  call initializeRoots(V, par)
  call saturateSourceSink(A, c, s, t)
  call setBranchStatus(V_I, f)

procedure initializeRoots(V, par):
  foreach node v ∈ V:
    par(v) ← nil

procedure saturateSourceSink(A, c, s, t):
  foreach arc (u, v) ∈ A:
    if u = s or v = t:
      f(u, v) ← c(u, v)
    else:
      f(u, v) ← 0

procedure setBranchStatus(V_I, f):
  foreach root node v ∈ V_I:
    if e_f(v) > 0:
      mark v strong
    else:
      mark v weak
```

The simple initialization procedure creates initial positive excess at source-adjacent nodes. All of this positive excess is the result of saturating the arcs out of the source. No other positive excess will be created during the execution of the pseudoflow algorithm. This initial excess will only be pushed through the graph during the rest of the pseudoflow algorithm. Therefore, we can say that all of the positive excess originates at the source.

Similarly, negative excess is created adjacent to the sink, and this can be said to originate from the sink. However, negative excess will not move during the execution of the algorithm—i.e., the negative excess will remain at the sink-adjacent nodes.

2.2.2 Phase I of the Generic Pseudoflow Algorithm

After an initialization procedure establishes an initial pseudoflow f and the initial parent-child relationship, par(v), Phase I of the algorithm only operates on the nodes and edges of G_I—i.e., Phase I ignores the source and sink nodes, and the arcs to and from the source and sink.

A *merger arc* (v, w) is an arc with positive residual capacity, $r_f(v, w) > 0$, between a strong node v and a weak node w.

Each iteration of the algorithm begins by identifying a merger arc. We denote the strong node v and the weak node w. Let the root of the strong branch be r_v, and let the root of the weak branch be r_w.

If v is not the root of the strong branch, we *rehang*[2] the strong branch from v. This involves reversing the parent-child relationships of all nodes on the path from v to r_v. That is, for a parent node u with child node v, v will become the parent of u. When complete, v is the root of the branch, but we will continue to refer to the original root as r_v.

Note that this only affects the nodes along the path from v to r_v—i.e., if u is on the path to r_v, and z is a child of u, but z is not on the path, then after rehanging the branch, u is still the parent of z. See Figure 2.1.

The pseudocode to rehang a branch from a node v is shown below. When the procedure returns, v is the root of the branch.

```
procedure rehang(v, par):
  p ← par(v)
  if p ≠ nil:
    call rehang(p, par)
    par(p) ← v
    par(v) ← nil
```

After rehanging the strong branch from the strong node v, we continue the merger by attaching v under w, which merges the two branches to create a single branch in which v is a child of w. This new single branch violates the fourth normalized forest property in

[2] In the literature, this operation is often referred to as *inverting* a branch or tree. Sleator and Tarjan [ST83] refer to this operation as *everting* a tree.

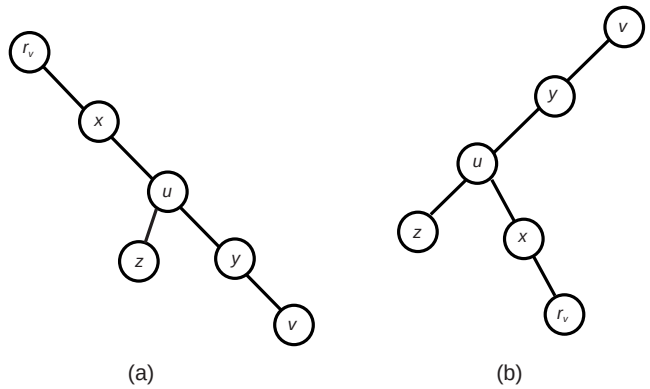


Figure 2.1: Example of a rehang operation. (a) The root of the strong branch is r_v, and the strong merger node is v. (b) After rehanging the strong branch from v, the parent-child relationships between all nodes along the path from r_v to v are reversed. Note the children of u: after rehanging, z is still the left child, but now x is the right child, instead of y.

that there is still positive excess at r_v, which is no longer the root of a branch. Therefore we need to *push* the excess from r_v to r_w. For each node along the path, this involves sending the excess from the node to its parent. We call the portion of the push from r_v to v a *strong push* and the portion from w to r_w a *weak push*.

If the amount of the excess at node i exceeds the residual capacity of the arc (i, j) to its parent j, the branch is *split* creating a new strong branch with root i and initial excess equal to $e_f(i) - r_f(i, j)$. The amount of excess ($r_f(i, j)$) that can be pushed across the arc to j continues to be pushed along the path towards r_w.

Because all downwards residual capacities are positive (by the third property of normalized forest), and the merger arc has positive residual capacity, a positive amount of excess reaches the weak branch. If the residual capacities of all arcs along the path from w to r_w are positive, then a positive amount of excess reaches r_w. If the amount of excess to reach r_w exceeds the deficit at r_w, the branch becomes strong; otherwise, the branch remains weak. Otherwise, if an arc (i, j) along the path from w to r_w has zero residual capacity, no additional excess will reach r_w; however, a new strong branch will be created with root i.

The pseudocode for this entire process of pushing excess from a node u to the root of its branch performing the splits is shown below.

```
procedure renormalizePath(u, f, par):
  δ ← e_f(u)
  while par(u) ≠ nil:
    v ← par(u)
    δ ← min(r_f(u, v), δ)
    f(u, v) ← f(u, v)+δ
    if e_f(u) > 0:
      ⟨⟨excess exceeded residual capacity—split u from v⟩⟩
      par(u) ← nil
      mark u as strong
    u ← v
  end while

  if e_f(u) > 0:
    mark u as strong
```

Note that the merger process removes one strong branch (the original branch involved in the merger) and creates zero or more new strong branches. The number of weak branches either remains unchanged or decreases by one if the weak branch becomes strong.

Let *genericInit* denote any initialization function that establishes an initial pseudoflow and a normalized forest in accordance with the properties of normalized forest. The complete pseudocode for Phase I and initialization is fairly simple:

```
procedure genericPseudoflow(V, A, s, t, c):
  call genericInit(V, A, s, t, c)
  while there is a merger arc (v, w):
    call merge(v, w, f, par)

procedure merge(v, w, f, par):
  call rehang(v, par)
  par(v) ← w        ⟨⟨attach v as child of w⟩⟩
  call renormalizePath(r_v, f, par)
```

Each iteration in Phase I consists of identifying a merger arc and performing a merger. The merger process moves positive excess from the root of the strong branch to the weak branch. This continues until no merger arc can be found, at which point, there are no arcs in the residual graph from a strong node to a weak one.[3] The pseudoflow may not

[3]Note that the status of a node can alternate between strong and weak many times during the execution of the algorithm.

be a feasible flow because the flow balance constraints for the branch roots may still be violated.

Hochbaum [Hoc97] proved the following lemmas and theorems:

Lemma 2.2.1 *Each iteration of Phase I of the generic pseudoflow algorithm either reduces the total excess of the strong nodes, or it makes at least one weak node strong.*

Let M^+ denote the capacity of the arcs out of the source—i.e., $M^+ = \sum_{(s,v)\in A} c(s,v)$.

Theorem 2.2.2 *For integer capacities, Phase I of the generic pseudoflow algorithm requires $O(nM^+)$ iterations.*

Note that this complexity is pseudo-polynomial because it depends on M^+. Later, we present a strongly polynomial algorithm, so we are not very motivated to search for the most efficient implementation of each iteration of this algorithm. Clearly, we can identify a merger arc by scanning every arc in the graph in time $O(m)$. The work of the merger requires pushing the flow at most n steps along the merger path. Therefore the work per merger is $O(m+n)$, and hence, the total complexity for Phase I of the generic pseudoflow algorithm is no worse than $O(nmM^+)$.

2.2.3 Phase II—Flow Recovery

Phase II converts the pseudoflow from Phase I into a feasible flow with maximum value—i.e., a maximum flow. We call this process *flow recovery*. This flow recovery procedure can be applied to the pseudoflow resulting from any Phase I algorithm—i.e., it is not limited to the Phase I procedure for the generic pseudoflow algorithm shown above. Flow recovery returns excess to the source and deficit to the sink. Thus, we will be considering the entire graph including the source and sink.

Given a network $G = (V, A)$ with source s and sink t and a pseudoflow f generated by Phase I of a pseudoflow algorithm, let S be the set of strong nodes at the end of Phase I plus the source s. Let $\overline{S}$ be the nodes in $V \setminus S$. (In other words, $\overline{S}$ is the set of weak nodes and the sink t.) $(S, \overline{S})$ is the set of arcs between S and $\overline{S}$.

The original positive excess in the graph resulted from saturating the arcs out of the source during initialization. Part of this original excess has been moved to the weak nodes

through the arcs in $(S, \overline{S})$ so that the remaining positive excess in S is less than the capacities of the arcs out of the source. We can use flow decomposition on the pseudoflow to determine a series of paths with positive flow from the source to each node with positive excess[4] and reduce the flow along the paths to return all of the excess to the source.

For every node v with positive excess with respect to f, we use a depth-first search to find a path in G from the source to v such that each arc in the path has a positive flow value, and we reduce the flow values along the path. Alternatively, we may identify a cycle containing v, in which case the flow along the cycle is reduced. We then send as much of the excess along this path/cycle as we can. The amount sent is the lesser of the excess of the node and the bottleneck capacity of the path/cycle. This returns all of the excess, or it saturates one or more arcs on the path/cycle. If the node still has excess, we search for more paths until the excess of the node is zero.

procedure returnToSource(v, V, A, s, f)**:**
  while $e_f(v) > 0$**:**
    let P be a path from s to v or a cycle containing v, s.t. $f(i,j) > 0 \; \forall \; (i,j) \in P$
    $\delta \leftarrow \min\{e_f(v), \min\{f(i,j) : (i,j) \in P\}\}$
    $f(i,j) \leftarrow f(i,j) - \delta, \forall \; (i,j) \in P$

The original negative excess in the graph resulted from saturating the arcs into the sink during initialization. This original deficit has been partially reduced by the positive excess of the strong nodes. The remaining negative excess is less that the capacities of the arcs into the sink. We can also use flow decomposition to identify paths from deficit nodes back to the sink. We use a function called *returnToSink*, which is nearly identical to *returnToSource*, except that it searches for from the node to the sink.[5]

procedure returnToSink(v, V, A, t, f)**:**
  while $e_f(v) < 0$**:**
    let P be a path from v to t or a cycle containing v, s.t. $f(i,j) > 0 \; \forall \; (i,j) \in P$
    $\delta \leftarrow \min\{-e_f(v), \min\{f(i,j) : (i,j) \in P\}\}$
    $f(i,j) \leftarrow f(i,j) - \delta, \forall \; (i,j) \in P$

[4]The nodes with positive excess during Phase II were strong root nodes at the end of Phase I. Similarly, nodes with negative excess at the end of Phase I were weak roots. In Phase II, we are not concerned with the branch structure, so we merely refer to nodes with non-zero excess.

[5]Since most initialization schemes (including simple initialization) only create deficit nodes adjacent to the sink, we can alternatively reduce the flows on the arcs from the deficit nodes into the sink. The amount of deficit at a node must be less than or equal to the capacity of the arc from the node to the sink.

To recover the flow, we repeat these steps for each node in V_I with nonzero excess. At this point, the flow balance constraints are met for all nodes in the graph except the source and sink, and the capacity constraints are met for all arcs in the graph—i.e., the flow is feasible. The complete flow recovery procedure is shown below.

procedure flowRecovery(V, A, s, t, f)**:**
  foreach node $v \in V_I$ such that $e_f(v) \neq 0$**:**
    if $e_f(v) > 0$**:**
      call returnToSource(v, V, A, s, f)
    else if $e_f(v) < 0$:
      call returnToSink(v, V, A, t, f)

Hochbaum [Hoc97] proved the following theorems related to flow recovery:

Theorem 2.2.3 *Phase II constructs a feasible flow in time* $O(m \log n)$.

Theorem 2.2.4 *The flow generated by the complete pseudoflow algorithm is a maximum flow.*

With these and the results of Phase I, we can see that the complexity for the entire generic pseudoflow algorithm is no worse than $O(nmM^+)$.

2.3 Lowest Label Algorithm

Phase I of the generic pseudoflow algorithm is pseudo-polynomial, which leaves room for improvement. In this section, we develop a new Phase I algorithm called the *lowest label* pseudoflow algorithm that is strongly polynomial. The generic pseudoflow algorithm does not specify how merger arcs are selected. We introduce a scheme for labeling the nodes in V_I and present an algorithm that uses the labels to select merger arcs.[6] This merger arc selection policy results in a strongly polynomial Phase I algorithm.

The labeling scheme is somewhat similar to the labeling scheme used in the push-relabel algorithm [GT88]. For each node v in V_I, we associate a positive integer called the *label* of the node. Let the label of a node $v \in V_I$ be denoted by $l(v)$.

[6]Note that this presentation combines the concepts described in Hochbaum [Hoc97] as the "lowest label" and "phase" variants of the pseudoflow algorithm.

Given a graph $G = (V, A)$, a pseudoflow f, and a normalized forest structure defined by the parent-child relationship *par*, for all nodes $v \in V_I$, the labels $l(v)$ satisfy the following properties throughout the execution of the lowest label Phase I algorithm [Hoc97]:

Property 1: For all arcs (u, v) in the residual graph G_f, $\ell(u) \leq \ell(v) + 1$.

Property 2: [Monotonicity] Within a branch defined by *par*, along any path leading from the root downwards, labels of the nodes are nondecreasing.

Property 3: The label of a node is a lower bound on its distance to the sink in the residual graph G_f.

Property 4: Labels of nodes are nondecreasing over the execution of the algorithm.

Briefly, the lowest label algorithm assigns initial labels to the nodes, and it selects merger arcs such that the labels of the strong and weak nodes are minimal. If it cannot find a merger between a given strong node and its neighbors, the label of the strong node is increased via a process called *relabeling*.

As is the case with constructing an initial normalized forest, there are numerous possible initial labeling schemes. The simplest scheme is to set the labels of all weak nodes to one and all strong nodes to two. We call this *constant labeling*. Later, we will present a more complex scheme based on distances to the nearest deficit node in the residual graph.

Each iteration of the lowest label algorithm begins by selecting the strong branch that contains strong nodes with the lowest label. Suppose the lowest-labeled strong node in the branch has label ℓ. We only visit the nodes within this branch that have label ℓ. Due to the monotonicity property, we know that the lowest-labeled nodes are all in the "top" portion of the branch.

For these strong nodes, we scan their neighbors in the residual network looking for a neighbor with a label less than ℓ. If we find such a node, it must be weak because the strong nodes we are visiting with label ℓ are, by definition, the lowest-labeled strong nodes. In fact, property 1 says the minimum value that the label of a neighbor (in the residual graph) can have is $\ell - 1$. Therefore, we only need to look for neighbors with label $\ell - 1$.

If we find a merger arc (v, w), the merger proceeds as it does in the generic pseudoflow algorithm: we rehang the strong branch from v, merge the branches by attaching v to w,

and push the excess from r_v to r_w, splitting as many times as needed.

To visit the nodes labeled ℓ within a branch, we use a depth-first, post-order tree traversal [CLR90]. This will visit the nodes in the branch labeled ℓ from the "bottom up." For a node v labeled ℓ, we visit all of its children labeled ℓ, searching for a merger. If we find no merger among the children of v, we search for a merger with the neighbors of v in the residual graph labeled $\ell - 1$. If we find no merger among the neighbors and children of v, we increase the label of v to $\ell + 1$, which indicates that v and its descendants have no neighbors in the residual graph labeled $\ell - 1$ or less. Because this process is a recursive, post-order traversal of the branch, we do not increase the label of a node to $\ell + 1$ until all of the descendants are labeled $\ell + 1$ or higher, which maintains monotonicity.

Therefore, each iteration either performs a merger between a node labeled ℓ and one labeled $\ell - 1$, or it increases the labels of all nodes in the strong branch labeled ℓ—from ℓ to $\ell + 1$.

The *processSubTree* function below recursively visits a subtree rooted at a strong node v searching for mergers and relabeling nodes if no merger arc is found. Other inputs to the function include the current pseudoflow$f$, the graph $G = (V, A)$, the parent-child relationship *par*, and the current node labels l. If the function performs a merger, it returns *true*; otherwise it returns *false*.

```
function processSubTree(v, f, V, A, l, par):
  ℓ ← l(v)
  foreach child c of v:
    if l(c) = ℓ:
      merged ← processSubTree(c, f, V, A, l, par)
      if merged:
        return true
  end foreach
  foreach neighbor w of v in G_f:
    if l(w) = ℓ − 1 and r_f(v, w) > 0:
      call merge(v, w, f, par)
      return true
  end foreach
  l(v) ← l(v)+1
  return false
```

Due to Lemma 2.3.3 below, we only need to scan the neighbors and children of a

strong node once while it has a given label value ℓ. Therefore, for each node we maintain a *pointer* into the lists of neighbors and children. These pointers are reset when the label of the node is increased, which indicates that it has no neighbor with a label less than ℓ.

With this function in place, the main loop of the lowest label algorithm is straightforward. Let *genericLabeling* denote a function that assigns an initial labeling to the nodes that satisfies properties one through three. The algorithm constructs an initial normalized forest and assigns initial labels to the nodes. The main loop selects a strong branch with the lowest-labeled root node and uses *processSubTree* to visit the nodes in the tree looking for mergers. It follows from Property 3 that no node label can exceed n, so we can stop the main loop if the root of the lowest-labeled strong branch has a label greater than or equal to n.

procedure lowestLabel(V, A, s, t, c)**:**
  call genericInit(V, A, s, t, c)
  call genericLabeling(V, A, s, t, f)
  while not done**:**
    select strong branch S with lowest-labeled root
    $r \leftarrow$ root of S
    if $l(r) > n$**:**
      done
    call processSubTree($r, f, V, A, l,$ par)

Hochbaum [Hoc97] proved that the labels satisfy the properties throughout the execution of the lowest label algorithm. Hochbaum also stated that the algorithm is correct because it only differs from the generic algorithm in the order of performing mergers.

Hochbaum also proved the following lemmas and corollaries related to the complexity of the lowest label algorithm:

Lemma 2.3.1 *Between two consecutive mergers using merger arc (v, w), the labels of v and w must increase by one unit each.*

Corollary 2.3.2 *Phase I of the lowest label pseudoflow algorithm executes at most $O(mn)$ mergers.*

Let us define *phase* number ℓ as the entire time we spend processing strong nodes with label ℓ.

Lemma 2.3.3 *All arcs incident to strong nodes in phase ℓ are traversed $O(1)$ times during the phase.*

Lemma 2.3.3 implies that the total work spent looking for mergers throughout the execution of the algorithm is $O(mn)$. In a straightforward implementation of the algorithm, the complexity of a merger is $O(n)$. However, this can be improved by using the dynamic trees data structure of Sleator and Tarjan [ST83] to reduce the complexity of each merger to $O(\log n)$.

Lemma 2.3.4 *[Hoc97] The complexity of the lowest label Phase I pseudoflow algorithm is $O(mn \log n)$.*

Thus, the lowest label Phase I algorithm is strongly polynomial. The flow recovery procedure (Phase II) is also strongly polynomial. Therefore, the complete maximum flow algorithm using the lowest label procedure is strongly polynomial.

2.4 Highest Label Algorithm

Let a *label-based algorithm* be a Phase I pseudoflow algorithm that searches for a merger from a node with label ℓ to one labeled $\ell - 1$. Hochbaum [Hoc97] observed that for a label-based algorithm to maintain the label properties above and to be of the complexity stated, it is not necessary to choose the lowest-labeled strong node among *all* strong nodes. It is sufficient that the node have the lowest label within any particular strong branch.

The *highest label* pseudoflow algorithm is a new label-based algorithm. It is motivated by the highest label variant of the push-relabel algorithm which processes the active node with the highest label [GT88]. Goldberg and Cherkassky [CG95b] indicated that the highest label variant is superior for "long and narrow" graphs because it tends to create many gaps that the gap relabeling heuristic of Derigs and Meier [DM89] can exploit.

The highest label pseudoflow algorithm is almost identical to the lowest label algorithm, and it maintains the same properties. It differs in that it chooses the strong branch with the *highest* root label rather than the lowest root label.[7] Once a strong branch is chosen,

[7] As the explanation suggests, a more accurate name for this algorithm might be the "highest root label algorithm."

it is processed in the same manner: we search down from the root visiting the nodes with the same label as the root ℓ. We search for a merger arc to a node labeled $\ell - 1$.

If we identify a merger between a strong node with label ℓ and another node labeled $\ell - 1$, we may merge with another strong branch because the label can no longer be used to differentiate between weak and strong nodes.[8] However, such a merger does not affect the correctness of the algorithm.

If there is no merger, we relabel the nodes that we scanned and continue processing. Note that this implies that we will select the same strong branch again because it was the highest labeled root before, and we just increased the label of its root node. Once the label of the root reaches n, we stop processing the branch.

The pseudocode for the highest label algorithm is shown below. It uses the same *processSubTree* function that is used by the lowest label algorithm.

procedure highestLabel(V, A, s, t, c)**:**
  call genericInit(V, A, s, t, c)
  call genericLabeling(V, A, s, t, f)
  while there exists a strong branch with root label less than n**:**
    select strong branch S with the highest labeled root
    let r be the root of S
    call processSubTree(r, f, V, A, l, par)

The label properties of the lowest label algorithm also apply to the highest label algorithm and can be proved in much the same way. Hochbaum [Hoc97] showed that the highest label algorithm has the same complexity as the lowest label algorithm. Although the complexity of the lowest and highest label algorithms is the same, in Chapter 3, we show that highest label algorithm performs significantly better in practice.

2.5 Simplex Merger

Another variant of the pseudoflow algorithm presented in Hochbaum [Hoc97] involves modifying the merger procedure used in Phase I. The new merger procedure does not alter the complexity of the pseudoflow algorithm, but it results in an algorithm that is

[8] With the lowest label algorithm, by definition if the lowest-labeled strong node has label ℓ, any node with label $\ell - 1$ must be weak.

similar to the network simplex algorithm. This modified merger procedure can be applied to any of the Phase I algorithms discussed above: generic, lowest label, or highest label.

Given a graph $G = (V, A)$, an initialization procedure establishes an initial pseudoflow f and normalized forest structure represented by *par*. A Phase I iteration begins, as usual, by identifying a merger arc (v, w) between a strong branch and a weak branch. We identify the arc with minimum residual capacity on the path $[r_v, \ldots, v, w, \ldots, r_w]$. This is the *bottleneck arc*, and its capacity is called the *bottleneck capacity*. If there is more than one arc with the same minimum capacity, we select the first one—i.e., the one closest to r_v along the path $[r_v, \ldots, v, w, \ldots, r_w]$.

The simplex merger procedure differs from the previous (typical) merger procedure in two ways: first, it will perform at most one split along the path from r_v to r_w; second, only a subset (or none) of the strong branch will be rehung from the strong node v and attached under the weak node w, as opposed to attaching the entire strong branch under w.

There are a number of cases to consider. First, if the excess of r_v is less than or equal to the bottleneck capacity, the merger is the same as the previous merger procedure, but note that there is no split along the path $[r_v, \ldots, r_w]$, and the strong branch becomes a subtree of the weak branch.

Otherwise, if the strong excess is greater than the bottleneck capacity, then we can only push the amount of excess equal to the bottleneck capacity. This implies that a positive amount of excess remains at r_v. In fact, r_v continues to be a strong root with positive excess, and we do not rehang the entire strong branch.

Let us assume that the bottleneck arc is (i, j), where i is closer to r_v, and j is closer to r_w. We only push $\delta = r_f(i, j)$ units of excess along the path $[r_v \ldots r_w]$. There are three possibilities to consider with regard to the location of the bottleneck arc along the path from r_v to r_w.

1. The bottleneck arc lies in the strong branch: (i, j) is somewhere along the path from r_v to v. The portion of the strong branch from r_v down to i will continue to be strong. We pull δ units of excess down from the root, r_v. The arc (i, j) is saturated by increasing the flow by δ units, and j is split from i—i.e., i is no longer the parent

Figure 2.2: Simplex merger when the bottleneck arc (i, j) is within the strong branch. After the merger (b), the portion of the strong branch above i remains strong, and the rest becomes weak and is attached under w.

of j. The portion of the branch from j down to v becomes weak. This portion of the strong branch is rehung from v and attached to w as a child. The excess δ is pushed from j to r_w, without generating any further splits. This is shown in Figure 2.2.

2. The bottleneck arc is the merger arc—i.e., (i, j) is the same as (v, w) in our typical notation for mergers. In this case, no rehanging is necessary. We pull δ units of excess down from r_v to i, and we increase the flow on (i, j) by δ units, thus saturating (i, j). Finally, we push the excess from j up to r_w without any additional splits.

3. The bottleneck arc lies in the weak branch: (i, j) is somewhere along the path from w to r_w. In this case, the portion of the weak branch from w to i becomes part of the strong branch. To do this, we split i from j (j is no longer the parent of i) and rehang the portion of the weak branch below i from w and attach w as a child under v. Now, we pull δ units of excess from r_v down to i and increase the flow on (i, j) by δ units, saturating it. Finally, we push the excess from j up to r_w without any additional splits. This is shown in Figure 2.3 below.

   Notice that this is very different from the typical pseudoflow algorithm merger process. Normally, the strong branch is attached to the weak branch. However, in this case a portion of the weak branch is attached under the strong branch, and that portion of the weak branch is rehung instead of the strong branch being rehung.

To summarize the procedure: if the excess of the strong root $e_f(r_v)$ is less than or equal to bottleneck capacity $\delta = r_f(i, j)$, the merger is performed using the standard procedure presented earlier. Otherwise ($e_f(r_v) > \delta$), δ units of excess are pushed along

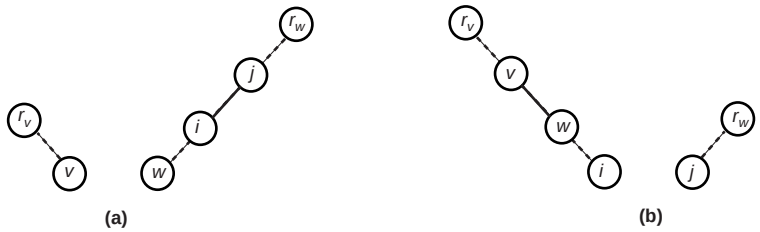


Figure 2.3: Simplex merger when the bottleneck arc (i, j) is in the weak branch. (b) The portion of the weak branch below j is rehung from w and attached under v.

the path $[r_v, \ldots, v, w, \ldots, r_w]$. The nodes along the path from r_v to i will be strong after the merger, the nodes along the path from j to r_w will be weak, and the bottleneck arc (i, j) will be saturated and removed from the normalized forest.

The pseudocode for the simplex merger procedure is shown below:

procedure simplexMerge(v, w, f, par)**:**
  let (i, j) be the bottleneck arc along the path $[r_v, \ldots, v, w, \ldots r_w]$
  $\delta \leftarrow r_f(i, j)$
  if $e_f(r_v) \leq \delta$**:**
    call merge(v, w, f par)
  else:
    foreach arc (i, j) on the path $[r_v, \ldots, v, w, \ldots, r_w]$**:**
      $f(i, j) \leftarrow f(i, j)+\delta$
    end foreach
    if $(i, j) \in [r_v \ldots v]$**:**
      par(j) $\leftarrow$ nil $\langle\langle$*remove j as child of i*$\rangle\rangle$
      call rehang(v, par)
      par(v) $\leftarrow w$ $\langle\langle$*attach v as child of w*$\rangle\rangle$
    else if $(i, j) \in [r_w \ldots w]$:
      par(i) $\leftarrow$ nil $\langle\langle$*remove i as child of j*$\rangle\rangle$
      call rehang(w, par)
      par(w) $\leftarrow v$ $\langle\langle$*attach w as child of v*$\rangle\rangle$
    else:
      $\langle\langle(i, j)$ *is the merger arc—no need to modify branch structure*$\rangle\rangle$

This simplex merger can be applied to the generic, lowest label, or highest label pseudoflow algorithms. When it is applied to the lowest or highest label algorithms, there is a minor problem. In the third case above, we attach a portion of the weak branch under the strong branch. However, with lowest and highest label algorithms, the label of the weak node, w, is less than that of the strong node, v. Furthermore, the labels along the path from w

to r_w are nonincreasing.

This means that when the merger is complete, r_v is still the root of the branch; however, r_v is not the lowest-labeled node in the branch. The subtree below v contains nodes with labels less than the label of r_v—i.e., the branch no longer has the monotonicity property. We call such a branch *degenerate.* This conflicts with our implementation that assumes monotonicity of the labels within strong branches.

The solution is to maintain our basic technique for processing branches—visiting the lowest-labeled nodes first—even though the lowest-labeled node is no longer the root of the branch. This is a significant, but not insurmountable, complication for the implementation.

As we visit the lowest-labeled portions of the degenerate branch, if we do not find a merger, these nodes are relabeled, and eventually monotonicity is restored to the branch. If there is a merger, we again have to consider the above three cases. If a portion of this branch is rehung and attached to the weak node in case 1, then monotonicity is restored for that rehung portion of the branch. Any portion that was degenerate before the merger and that was not rehung, will continue to be degenerate and will continue to need this special handling until monotonicity is restored.

During a merger that initially creates a degenerate branch, we flag the nodes that are in the portion of the branch that is degenerate. Subsequently, when we process the branch with *processSubTree*, we start the processing from the lowest portion of the branch that is still degenerate. By noting the degeneracy during the merger when we know exactly which portions are degenerate (those nodes under v) rather than waiting until we process the strong branch again, we avoid the need to search the entire branch looking for the lowest labeled nodes.

Therefore, these degenerate branches do not affect the complexity of the simplex algorithm. The search for a merger arc (performed by *processSubTree*) is unchanged except that it may initially start from a node that is not the root of a branch. Mergers still occur between nodes labeled ℓ and those labeled $\ell - 1$. Before a merger arc (v, w) can be used again as a merger arc, the labels of v and w must both increase by at least one. Therefore, the number of mergers for a label-based algorithm is still $O(mn)$.

Regardless of which variant of Phase I the simple merger procedure is applied to, the complexity of the Phase I algorithm is unchanged. As Hochbaum[Hoc97] observed, this merger procedure (when viewed in a transformed version of the graph) is very similar to the network simplex algorithm. Network simplex maintains a spanning tree where the flow on every arc not in the current spanning tree is either at its upper or lower capacity bound. Each iteration of the network simplex algorithm moves from one spanning tree solution to another by adding an out-of-tree arc, adjusting the flows, and removing an in-tree arc whose flow is at the upper or lower capacity bound [Dan63].

Given a graph $G = (V, A)$, a pseudoflow f, and a normalized forest structure defined by *par* resulting from the ongoing execution of a Phase I pseudoflow algorithm, we can construct an *extended network* [Hoc97] as follows. The source s and sink t are shrunk into a single "root" node r, and the network is augmented with a set of additional arcs to and from r. For each branch in the normalized forest, let v denote the root of the branch. For strong branches, we add an *excess arc* (r, v) from r to v with residual capacity equal to the excess of the strong branch. For weak branches, we add a *deficit arc* (v, r) from v to r with residual capacity equal to the deficit of the branch. We refer to the set of arcs in the extended network as A^x, where $A^x = \{(r, v) : v \text{ is root strong root}\} \cup \{(v, r) : v \text{ is root weak root}\} \cup \{(i, j) : (i, j) \in A,\ i \in V_I, \text{ and } j \in V_I\}$. The extended network is the graph $G^x = (V_I \cup \{r\}, A^x)$.

In this extended network, the normalized forest plus the root r form a single rooted spanning tree called a *normalized tree*. All arcs from the original graph which are not part of the normalized tree have zero flow or are saturated.

With the simplex merger procedure, the addition of the merger arc (v, w) to the normalized tree creates a cycle in the extended network $[r, r_v, \ldots, v, w, \ldots, r_w, r]$, just as the entering arc creates a cycle in network simplex. Flow is pushed along this cycle to saturate the bottleneck arc. When the bottleneck arc is removed, a new spanning tree is formed, just as removing the leaving arc does in network simplex. If the excess at the strong root $e_f(r_v)$ is less than the capacities of arcs long the path $[r_v, \ldots, r_w]$, then the excess arc (r, r_v) is removed—i.e., the excess arc is the leaving arc. Similarly, if the strong excess $e_f(r_v)$ and the capacities along the path $[r_v, \ldots, r_w]$ exceed the deficit at the weak

root r_w, then the deficit arc (r_w, r) is the leaving arc, and it is removed.

Thus, the algorithm begins each iteration with a spanning tree. The meger/entering arc forms a cycle along which flow is pushed to saturate the bottleneck arc, which becomes the leaving arc. Once the leaving arc is removed, a new spanning tree is formed.

2.6 Heuristics and Variants

One interesting feature of the pseudoflow algorithm is that there are many opportunities for heuristics and variations. In this section we present some of the heuristics that we investigated.

2.6.1 Initialization

The simple initialization scheme presented in Section 2.2.1 simply saturates the arcs out of the source and those into the sink. However, there are many more possibilities. All of the initialization schemes include saturating the source and sink arcs, as required by the properties of the normalized forest.

As noted before, the simple initialization scheme produces a normalized forest with single node (singleton) branches. Other initializations (described below) lead to larger branches. For all of the initialization schemes, the in-tree edges of the branches are defined by $E_T = \{[u, v] : \text{par}(u) = v\}$. The roots of the branches are noted for each initialization scheme.

2.6.1.1 Blocking Path

In the blocking path initialization method, after saturating the source and sink arcs, we push the excess at each source-adjacent node along a path as far into the graph as possible.

For each node v with positive excess, we scan its outgoing arcs in the original graph to find an arc (v, u) with sufficient capacity to handle the entire excess. If we find an arc with sufficient capacity, we make v a child of u and push the excess to u by setting the flow equal to the excess, $f(v, u) = e_f(v)$.

The process continues until the excess reaches a node that lacks an arc with sufficient capacity to accommodate the excess, or we reach a sink-adjacent node. If we cannot push the excess from the node, we stop and move onto the next node with positive excess. If we

push the excess to a sink-adjacent node, we sum the excess and deficit of the sink-adjacent node, thereby reducing the deficit or creating positive excess at the sink-adjacent node.

In the case where there are multiple arcs with sufficient capacity, we can choose any arc. In such a case, we will choose the first such arc in the node's list of out-arcs.

Finally, we set the status of the resulting branches as either strong or weak using *setBranchStatus* shown in Section 2.2.1.

procedure blockingPushInit(V, A, s, t, c)**:**
  call initializeRoots(V, par)
  call saturateSourceSink(A, c, s, t)
  mark all nodes unvisited
  foreach source-adjacent node u such that $e_f(u) > 0$**:**
    call blockingPush(u, f, c, par)
  call setBranchStatus(V_I, f)

procedure blockingPush(u, f, c, par)**:**
  mark u visited
  find the first arc (u, v) such that $c(u, v) > e_f(u)$ and v not visited
  if such an arc exists**:**
    par(u) $\leftarrow v$
    $f(u, v) \leftarrow e_f(u)$
    blockingPush(v, f, c, par)
  end

This process creates branches that are paths. The number of branches/paths equals the original number of strong nodes. The roots of the strong branches are the nodes at which no arc of sufficient capacity could be found. The leaf nodes are the source-adjacent nodes that contained the original excess. The weak branches are singletons. To insure that we do not create any cycles, we mark each node as visited when we push excess through the node. The complexity for the entire initialization process is $O(m)$.

2.6.1.2 Greedy Initialization

When the blocking path initialization method encounters a node with positive excess that has no single arc with sufficient capacity, the process stops at that node. A simple, obvious extension is to allow flow out on multiple arcs. If no arc out of node u has sufficient capacity for the excess $e_f(u)$, we saturate the arc (u, v) with maximum capacity, but we do not add u as a child v. Now we have two nodes, u and v, with positive excess. Each

of these is the root of a separate branch with excess at the root (v is a singleton branch). Each branch can be processed independently, searching to push flow far away from the source.

procedure greedyInit(V, A, s, t, c)**:**
  call initializeRoots(V, par)
  call saturateSourceSink(A, c, s, t)
  mark all nodes unvisited
  foreach source-adjacent node u such that $e_f(u) > 0$**:**
    call greedyPush(u, f, c)
  call setBranchStatus(V_I, f)

procedure greedyPush(u, f, c, par)**:**
  mark u visited
  find arc (u, v) with maximum capacity such that v not visited
  if $c(u, v) > e_f(u)$**:**
    par$(u) \leftarrow v$
    $f(u, v) \leftarrow e_f(u)$
    call greedyPush(v, f, c, par)
  else:
    $f(u, v) \leftarrow c(u, v)$
    call greedyPush(v, f, c)
    call greedyPush(u, f, c)

As was the case with blocking path initialization, we avoid cycles by marking nodes as we visit them. This prevents pushing excess through the same node twice. The result is also a series of branches that are paths, but not all leaf nodes in the branches are adjacent to the source, as was the case with blocking path initialization. The weak branches are singletons. The complexity is $O(m)$.

Obviously, this may generate more branches than the original number of strong nodes. This method can also push the flow further into the graph than the blocking path method since it can push excess out of a node through multiple arcs. For example, many synthetic problem classes have arcs out of the source with capacity far greater than the outgoing capacity of the source-adjacent node—i.e., $c(s, u) > \sum_{(u,v)\in A} c(u, v)$. With the blocking path initialization, the excess cannot leave the source-adjacent node, but with greedy initialization the excess can be pushed further into the graph.

2.6.1.3 Shortest Path Initialization from Sink

The previous two initialization schemes grow strong branches (paths) out from the source-adjacent nodes while pushing excess into the graph. In this section, we build weak branches from the sink-adjacent nodes. One way to do this is to build branches such that for every node in the branch, the parent of the node is one arc closer to the sink (in the residual graph) than the node. This creates a structure that is inherently acyclic.

We begin by saturating the arcs out of the source and those into the sink. To establish the distances to the sink in the residual graph, we perform a breadth-first search backward from the sink-adjacent nodes in the residual graph. Initially, we set the distances of all nodes to infinity. Then we set the distances of the sink-adjacent nodes to one and insert them into a FIFO queue. The breadth-first search proceeds by removing a node u from the queue and scanning its zero-deficit neighbors. If the residual capacity from another node v into u is positive ($r_f(v, u) > 0$), and we have not already visited v (i.e., its distance is still infinity), we set the distance of v to be $d(u) + 1$, make v a child of u, and add v to the queue. This continues until we have visited all the zero-deficit nodes in the graph and the queue becomes empty.

procedure shortestPathInit(V, A, s, t, c)**:**
  call initializeRoots(V, par)
  call saturateSourceSink(A, c, s, t)
  $\forall v \in V_I$, $d(v) \leftarrow \infty$
  initialize empty queue
  foreach sink-adjacent node v such that $e_f(v) < 0$**:**
    d(v) $\leftarrow 1$
    add v to queue
  while queue not empty**:**
    $u \leftarrow$ first node in queue
    $d \leftarrow$ d(u) +1
    foreach node v adjacent to u**:**
      if $r_f(v, u) > 0$ and $e_f(v) = 0$ and $d(v) = \infty$**:**
        $d(v) \leftarrow d$
        par(v) $\leftarrow u$
        put v in queue
    end foreach
  end while
  call setBranchStatus(V_I, f)

This procedure creates strong nodes that are singleton branches adjacent to the source. The roots of the weak branches are the sink-adjacent nodes. All nodes with zero deficit belong to one of these weak branches. The flows on the edges in the trees are all zero.

The complexity of the procedure is $O(m)$. The resulting branches are not simple paths: each node may have multiple children. Note that this technique of growing a branch structure with zero flows on the arcs from the sink could be adapted to grow strong branches out from the source. However, we did not implement this.

2.6.1.4 Saturate-All Initialization

In this initialization, we saturate all arcs in the graph. For each node, the net excess is the sum of the inflows and outflows.

procedure saturateAllInit(V, A, s, t, c)**:**
  call initializeRoots(V, par)
  foreach arc $(u, v) \in A$**:**
    $f(u, v) \leftarrow c(u, v)$
  call setBranchStatus(V_I, f)

This procedure produces singleton branches. Nodes with positive excess are strong, and nodes with non-positive excess are weak. In contrast to the other initialization methods we have presented, the saturate-all initialization may create deficit nodes that are not adjacent to the sink.

2.6.2 Label Gaps

Once initialization is complete, the label-based pseudoflow algorithms proceed by visiting the nodes in strong branches. While visiting strong nodes labeled ℓ with either the lowest label or highest label algorithm, if there are no nodes labeled $\ell - 1$ we say that there is a *gap* in the labels. In this section we present heuristics to reduce the running time of an implementation of the pseudoflow algorithm when a gap is detected.

2.6.2.1 Lowest Label Algorithm

Recall that because the label of a node cannot exceed n, we terminate Phase I of the lowest label algorithm when the lowest-labeled strong node reaches n. However, very early during the implementation presented in the next chapter, we noticed that the end of Phase I was marked by many relabel operations without any mergers. The labels of the strong nodes were simply rising to n.

A heuristic for the push-relabel algorithm that was discovered early in the development of the push-relabel algorithm is the gap relabeling heuristic of Derigs and Meier [DM89]. This inspired the following observation for the lowest label pseudoflow algorithm.

Lemma 2.6.1 *If the lowest-labeled strong branch has root label ℓ, and there are no nodes in the graph with label $\ell - 1$, then there is no path in the residual graph from a strong node to a weak node or to the sink, and Phase I of the lowest label algorithm can terminate.*

This follows immediately from Property 1. Any path in the residual graph from a node labeled ℓ to a node with label less than ℓ (e.g., a weak node or the sink) must pass through a node labeled $\ell - 1$.

In the push-relabel algorithm, the gap allows the algorithm to stop processing the subset of nodes that have labels greater than ℓ because they are in the source set of

the minimum cut. In the lowest label pseudoflow algorithm the gap allows us to stop processing *all* strong nodes and finish Phase I.

This simple heuristic is the most powerful one we have developed. In fact, it is a standard part of the implementation because there is no case where it performs poorly.

To implement this rule, we simply keep n counters, one for each possible node label value. The initialization procedures set the counters based on the initial number of nodes with each label value. When a strong node is relabeled from ℓ to $\ell+1$, the counter for ℓ is decreased by one, and the counter for $\ell+1$ is increased by one. When we begin processing a branch with root label ℓ, we simply consult the counter for $\ell-1$ to see if there are any nodes with label $\ell-1$. All operations of updating and checking the counter are executed in constant time.

2.6.2.2 Highest Label Algorithm

In the highest label algorithm, if a strong branch has root label ℓ, and there exist no nodes with label $\ell-1$, then the branch will not merge with any other branches in *processSubTree*, and the nodes in the branch labeled ℓ will be relabeled to $\ell+1$. At this point, there will be no nodes labeled $\ell-1$ or ℓ, and the procedure will continue until the strong branch is labeled n.

Therefore, we recognize that when there are no nodes labeled $\ell-1$, there will be no more mergers from this branch to any others. So, we can stop processing the branch. To do this we relabel the nodes in the branch to n and move on to the next highest labeled branch. We call this operation *pruning* the branch.

To implement the check for nodes labeled $\ell-1$, we use the same counters described above for the lowest label algorithm.

2.6.3 Distance-Based Labels

The label of a node can be viewed as a lower bound on the distance to a deficit node (weak root) in the residual network.

Let $d_w(v)$ denote one plus the distance (number of arcs) from a node v to nearest deficit node in the residual network. For a weak root r, $d_w(r) = 1$.

Lemma 2.6.2 *The label of a node v is a lower bound on its distance to the root of a weak branch in the residual network plus one—i.e., $l(v) \leq d_w(v)$.*

Proof: For a weak node, this is simply a restatement of Property 3. For other nodes in V_I, this follows from Property 1. ■

In the more general case where roots of weak branches may have labels greater than one, then $l(v) - l(r_w)$ is a lower bound on the distance between v and a weak root r_w in the residual graph. In our current implementation, weak roots always have labels equal to one. Hence, we use the form in Lemma 2.6.2.

To establish the distances to deficit nodes, we perform a breadth-first search in the reverse residual graph from the deficit nodes. We call this procedure *distance to deficit* labeling.

For most initialization schemes, the weak roots are adjacent to the sink.[9] In this case, we can modify the initialization step that loads the breadth-first search queue to only consider the sink-adjacent nodes, which may be faster in practice than scanning all the nodes in V_I to identify deficit nodes. These nodes all have a label of one, and they are one arc away from the sink. Therefore, the distance $d_w(v)$ represents the distance from v to t in the residual network. We call this special-case procedure *distance to sink* labeling.

In order to maintain the labeling properties, we cannot simply set the node labels to match the distance labels. In particular, we need to maintain the monotonicity property. If a node v is the closest node in a branch to a deficit node, and it is not the root of the branch, then its parent and other ancestors may have distance labels greater than the distance label of v. If the node labels were updated to match the distance labels, the labels of some parent nodes would exceed the labels of their children, which violates the monotonicity property.

Therefore, we regard the distance $d_w(v)$ as an upper bound on how high we can set the label of the node, $l(v)$. We process the nodes in a branch from the leaves up using a post-order traversal. If a node has no children, the label can be set to match the distance label. Otherwise, we update the labels of the children of v recursively, compute the minimum of

[9]The notable exception is the saturate-all initialization which may create deficit nodes that are not adjacent to the sink.

the new labels of the children, and set the label of v no higher than that minimum.

The pseudocode to update the labels of the nodes in a branch while respecting monotonicity is shown below:

procedure relabelBranch(v, l, par)**:**
  if v has no children**:**
    $l(v) \leftarrow d_w(v)$
  else:
    foreach child u of v**:**
      call relabelBranch(u, l,par)
    $\ell \leftarrow \min\{l(u) : u \text{ is a child of } v\}$
    $l(v) \leftarrow \min(\ell, d_w(v))$

We developed a pair of heuristics that aim to set or raise the label of a node using the distance to deficit information. The intent of these heuristics is to send excess toward deficit nodes via shortest paths in the residual graph.

2.6.3.1 Initial Labels

One way to use distance labeling is during initialization. After an initialization scheme has built branches and (optionally) pushed excess into the branches, we compute the distance to deficit labels and call *relabelBranch* on each branch to set the labels of the nodes. This can be used with any initialization scheme.

2.6.3.2 Global Relabeling

At any time during Phase I, it is possible to compute the distance to deficit labels with respect to the existing residual graph and raise the node labels to the match distance labels, subject to the monotonicity constraints within branches. We call this operation *global relabeling*, and it uses the distance to deficit and *relabelBranch* procedures.

Because it is a rather expensive operation, $O(m)$, global relabeling should not be performed too frequently. In the current implementation, the frequency of updates is controlled by the number of node relabel operations, although other schemes are possible. The algorithm takes a parameter g ($g \geq 0$) and performs a global relabeling operation after every gn node relabels (if $g = 0$, we do not perform relabeling). We focus on the number of relabels rather than mergers because Nguyen and Venkateswaran [NV93] indicated that it is a superior strategy for the push-relabel algorithm.

2.6.4 Branch Management

The roots of strong branches are stored in *buckets* based on the label of the root node; all strong branches with the same root label are stored in the same bucket. To identify a branch with the lowest-labeled strong nodes, we simply select a branch from the lowest-labeled bucket. Similarly, for the highest label algorithm, we select from the highest-labeled bucket. There are numerous ways to implement the buckets. Because the buckets can hold from zero to n branch roots, we chose to implement them as simple, singly-linked lists. We always remove branch roots from the head of the list, but we support three different schemes for inserting into the list:

LIFO: We add new branch roots to the head of the list. This makes the bucket a stack. Intuitively, LIFO should behave like a depth-first search. We start processing one strong node, and we continue pushing its excess as deeply into the graph as we can before the excess is absorbed by weak nodes.

FIFO: We add new branch roots to the tail of the list. This makes the bucket a queue. Intuitively, FIFO should behave like a breadth-first search while pushing excess out from the source. First, we process all the branches adjacent to the source. Then we process their neighbors one unit further from the source, and so on.

Wave: This is a combination of LIFO and FIFO. We usually put the branch root at the end of the list (FIFO) unless the root of the branch is the same root we last removed from a bucket; in this case, it is inserted at the head of the list (LIFO).

There are many other ways to manage the strong branches. One alternative would be to consider the excess of the root; for example, we could choose the strong branch with the most excess.

2.6.5 Search Order

When we process a strong node with label ℓ looking for a merger, we need both to scan the neighbors of the node and process each child labeled ℓ. However, we can apply these operations in either order—either scan the neighbors first and children second or vice versa.

Pre-order: We scan the node's neighbors before visiting the children.

Post-order: We visit the children first, and if there were are no mergers, we scan the neighbors.

Because the post-order procedure searches for merger arcs deeper in the strong branch (i.e., in child a subtree) rather than shallower (i.e., among the neighbors), we expect it would result in longer paths from a strong merger node to its root. This would result in more work for rehang operations and a longer path for the strong push. Therefore, we would expect post-order searches to run slower than pre-order searches.

2.6.6 Delayed Normalization

Usually after each merger between the strong node v and the weak node w, we renormalize the tree by pushing excess from the strong root r_v to v (strong push), across (v, w), and from w to the weak root r_w (weak push). We split edges that have insufficient residual capacity, thus creating new strong branches. This process restores the property that excess/deficit only exists at the root nodes of branches. However, we can temporarily suspend the property in the following manner: during a merger, we push the excess from the strong root as far as the weak node w, but we do not push it to the weak root r_w.

As we are processing strong branches with label ℓ, we leave the excess at the weak node. When we are done with strong branches labeled ℓ and are about to begin processing branches with a root label other than ℓ, we renormalize the tree by pushing the excess from weak nodes to their roots. That is, we complete the weak pushes that we deferred at the time of the merger.

Performing the strong push (rather than leaving the excess at the strong root, r_v) may create more strong branches with label ℓ, which we could continue to process. On the other hand, weak pushes create new strong branches with labels less than ℓ. By deferring the weak pushes, we delay changing to a lower phase in the lowest label algorithm.

Note that this is not the only way to delay normalization. One alternative would be to perform weak pushes every k mergers.

We only implemented delayed normalization for the lowest label pseudoflow solver, but it can also be used with the highest label algorithm.

2.6.7 Summary

In Table 2.1 we summarize the various options for the pseudoflow solver that are implemented and evaluated in this work.

Initialization	simple	saturate source and sink arcs
	path	blocking path from source
	greedy	greedy paths with splits
	shortest	shortest path weak branches
	saturate	saturate all arcs
Initial Label	const	strong nodes = 2, weak = 1
	sink	distance to sink
	deficit	distance to deficit node
Root Label & Normalization	lowest	lowest label, immediate normalization
	delayed	lowest label, delayed normalization
	highest	highest label, immediate normalization
Strong Branch Management	lifo	LIFO—stack
	fifo	FIFO—queue
	wave	FIFO—except for original strong branch
Search Order	pre	pre-order tree traversal
	post	post-order tree traversal
Global Relabel Period	0, 0.5, 1, 2, 4	relabel period as a factor of n (zero means no relabeling)
Merger	pseudo	standard, pseudoflow merger
	simplex	simplex merger

Table 2.1: Summary of heuristics and options for the pseudoflow solver.

We refer to a specific combination of heuristics by listing the name of each heuristic in the order that they appear in Table 2.1. For example, "simple-const-highest-lifo-pre-0" refers to simple initialization, constant initial labeling, highest-label algorithm, LIFO branch management, pre-order tree traversal, with no global relabeling. Note that unless stated otherwise, the merger type is pseudoflow.

Chapter 3

Experimental Results

As we saw in the previous chapter, the worst-case complexity of the pseudoflow algorithm is close to that of the push-relabel algorithm, although it is slightly worse. It is well known that the theoretical complexity of an algorithm and the actual performance of an implementation of the algorithm can be quite different.

In this chapter we investigate the performance of our first implementation of the pseudoflow algorithm compared to the best known implementation of the push-relabel algorithm as well as an implementation of Dinic's algorithm. We test the algorithms on a fairly large set of problem instances, including common synthetic problem instances and some instances from real world applications.

We also investigate the effects of the large number of heuristics for the pseudoflow algorithm. We select a pair of standard heuristic combinations to provide acceptable performance in most circumstances. We also identify the best heuristic combination for each problem family.

Our experiments show that the pseudoflow algorithm is quite competitive; in many cases the pseudoflow implementation is better than push-relabel, and even when it does not perform as well as push-relabel, its performance is typically close to that of push-relabel. This is a considerable accomplishment for the first implementation of a new algorithm compared to an algorithm that that has been implemented and studied extensively. The pseudoflow algorithm shows considerable promise. With additional effort, future implementations of the pseudoflow algorithm could even prove to be consistently superior.

3.1 Previous Maximum Flow Implementation Studies

In this section, we survey some previous implementations of maximum flow algorithms. In particular, recent studies have focused on the push-relabel algorithm and established it as the best algorithm in practice, which is why we primarily focus on comparing the pseudoflow algorithm to push-relabel later in the chapter.

3.1.1 Derigs and Meier—1989

The paper of Derigs and Meier [DM89] was one of the earliest to compare the performance of the push-relabel algorithm to Dinic's algorithm, which had previously been regarded as the best in practice. One of the most significant contributions of Derigs and Meier is the *gap relabeling* heuristic.[1] Given a maximum flow problem being solved by push-relabel and node labels $d(v)$, Derigs and Meier proved that if there exists a value $0 < \ell < n$ such that no node has label ℓ, then all nodes v such that $\ell < d(v) < n$ are disconnected from the sink in the residual graph and thus known to be in the source set of the minimum cut. Such nodes can be relabeled to $d(v) = n$.

The implementation of Derigs and Meier computes the exact distances from nodes to the sink in the residual graph to use as initial labels for the nodes. However, the implementation does not recompute the distances periodically during the execution. Therefore, it is not the same as the *global relabeling* heuristic[2] described by Goldberg and Tarjan [GT88], where the node labels are periodically updated throughout the execution of the algorithm to match their exact distance to the sink in the residual graph. In addition to the FIFO and LIFO variants of push-relabel, Derigs and Meier also implemented the highest label variant. All implementations were written in Fortran.[3]

The set of problem instances used for testing was fairly limited: they only used two classes of networks, NETGEN and RMFGEN. Derigs and Meier [DM89] concluded that push-relabel is superior to Dinic's algorithm. The highest label variant was generally the best, and "for the special structured RFMGEN-examples, the exact labeling seems to be

[1] Cherkassky [Che79] and Ahuja and Orlin [AO91] also discovered this heuristic independently.

[2] Derigs and Meier refer to the "gap relabeling" technique as "relabel-global," which is not the same thing as Goldberg and Tarjan mean when they say "global relabeling." In keeping with the terminology of most papers, we refer to the technique of Derigs and Meier as "gap relabeling."

[3] Unless noted otherwise, all other implementations described in this section were written in C.

essential" [DM89].

3.1.2 DIMACS Challenge—1991

Between November 1990 and August 1991, the First DIMACS Implementation Challenge was held to evaluate efficient implementations of minimum-cost flows, maximum flows, assignment, and non-bipartite matching problems. The challenge gathered a set of random graph generators and specified a methodology for testing implementations. This was used as the basis of performance comparisons during the challenge and subsequent papers, including ours. The proceedings were published in 1993 [JM93].

3.1.3 Anderson and Setubal—1993

Anderson and Setubal [AS93] compared the push-relabel algorithm to Dinic's algorithm and implemented older algorithms including Ford-Fulkerson, Karp and Edmonds, and Karzanov. They tested the implementations with a wide variety of input graphs.

The authors implemented four variants of the push-relabel algorithm: FIFO, LIFO, highest label, and largest excess. They also implemented both the gap relabeling and global relabeling heuristics individually, but they did not use both heuristics in the same program.

After initial testing on small instances, the authors identified push-relabel as superior to all other programs, except on acyclic dense graphs, where Dinic's algorithm was superior. They tested the four variants of push-relabel with larger graphs and found that the highest-label algorithm was generally superior to the other variants. The FIFO variant seemed to be "more robust, since it was never outperformed by large margins" [AS93]. They also found global relabeling to be typically better than gap relabeling, but did not test both heuristics in the same program. In fact, they found global relabeling can improve the performance by an order of magnitude in some graphs.

3.1.4 Nguyen and Venkateswaran—1993

Nguyen and Venkateswaran [NV93] implemented three variants of push-relabel: FIFO, LIFO, and highest label. They investigated the effects of global relabeling, and they also implemented the gap relabeling heuristic. However, like Anderson and Setubal [AS93],

they did not try the two heuristics together. The frequency used for global relabeling was based on the number of node relabel operations rather than the number of discharge operations used by Anderson and Setubal.

Nguyen and Venkateswaran confirmed that the highest label variant was superior for most instances, but that highest label is not the best for acyclic dense graphs. They stated, rather emphatically, that "global updates are crucial" [NV93], and that their strategy of performing global relabeling every n node relabels is superior to relabeling every $m/2$ discharge operations, which was used by Anderson and Setubal. However, this observation seems to be based on two separate implementations by the two sets of authors, rather than a single implementation with two separate policies. Therefore, some of the observed differences (especially in running times) may have been the result of coding differences in the two implementations.

3.1.5 Badics and Boros—1993

Badics and Boros [BB93] made aggressive use of theoretical techniques and data structures and compared the results to simpler implementations. They implemented two versions using dynamic trees [ST83]. The first implementation was based on the PLED (Prudent Linking and Excess Diminishing) variant of the push-relabel algorithm described by Cheriyan and Hagerup [CH89]. In addition to dynamic trees, this implementation used randomization of the edge lists and scaling of the amounts of flow pushed. The second version used dynamic trees in a straightforward implementation of the FIFO push-relabel algorithm. The authors also created versions of these algorithms using non-dynamic trees—i.e., they implemented the same tree operations using an explicit representation, and the operations were implemented with obvious linear-time code.[4] Finally, as a baseline, they also implemented a simple version of the FIFO algorithm. The authors implemented global relabeling based on the number of node selections (iterations) and the number of cuts in the tree structure. The code checks for gaps like the gap relabeling heuristic, but it uses the presence of a gap to initiate a global relabeling operation, rather than only relabeling the nodes with labels greater than the gap value.

[4]This is in contrast to dynamic trees that use an implicit representation of a forest. This results in $O(\log n)$ time for most operations.

In all cases, the simple FIFO implementation outperformed the more sophisticated implementations. For some problem instances where the flow could be pushed through long paths, the authors did find that the dynamic tree versions outperformed the versions that were implemented using non-dynamic trees. The latter showed exponential growth as a function of the number of nodes, while the dynamic trees maintained linear growth. The authors speculated that dynamic trees might be useful in graphs with a very long distance between the source and sink.

3.1.6 Goldberg and Cherkassky—1995

Goldberg and Cherkassky [CG95b] implemented Dinic's algorithm and several variants of push-relabel, but they only report on the highest-label and FIFO variants. The authors implemented both relabeling strategies because, as they point out, the two heuristics are not mutually exclusive.

They confirmed the results of previous authors that "the highest label variant of the push-relabel method with global and gap relabel heuristics is the best currently-available method for solving maximum flow problems" [CG95b]. Goldberg and Cherkassky had access to the implementations of Anderson and Setubal [AS93] and compared the performance of the implementations. The performance of the Goldberg and Cherkassky implementations were superior to those of Anderson and Setubal, so we used the Goldberg and Cherkassky implementations as the baseline for our experiments in in this Chapter.

3.1.7 Ahuja, Kodialam, Mishra, and Orlin—1997

Ahuja, Kodialam, Mishra, and Orlin [AKMO97] studied various algorithms with a slightly different emphasis. Rather than focusing exclusively on CPU time, they provide some in-depth analysis of the operation counts. They also chose to implement algorithms not studied by other authors, in particular variants of their own excess-scaling [AO89], stack-scaling [AOT89], and wave-scaling [AOT89] algorithms.

They chose to re-implement the highest label algorithm, rather than use an existing implementation. This may be related to their choice of Fortran as an implementation language, whereas nearly all recent implementations have been done in C. Also, they specifically chose not to implement global relabeling, despite the findings of almost all

previous authors that global relabeling is crucial. Furthermore, although they did some minimal testing with the DIMACS problem generators, they conducted the bulk of their work with other instances. Despite these issues, they found that the highest label push-relabel algorithm was superior to the other algorithms.

3.2 Software Implementations

In this section we discuss the software implementations we tested. This includes the pseudoflow, push-relabel, and Dinic's algorithms.

3.2.1 Pseudoflow Implementation

Our initial implementation of the pseudoflow algorithm is designed to be as flexible as possible to facilitate investigating different heuristics and variants. Later implementations can be tuned for speed based on these results.

We implemented the algorithm in C++ [ES90], and we followed an object-oriented design [Boo91] to maximize the flexibility and simplicity of the implementation.

The data structures we used are simple. Each node is represented by a C++ object that has a pointer to its parent (if any) and two lists of edges, one each for the neighbors and children. The list of neighbors contains edges for both arcs into and out of the node, and its size is fixed when the problem instance is initialized. The list of children is used as we build up trees, and its size is dynamic. It begins with zero entries when the node has no children and can grow up to the number of neighbors. These lists are implemented as simple arrays rather than linked lists or more sophisticated data structures. Pointers into the list are represented as simple integer indices.

Each edge in the original graph is represented as a single object containing the capacity and flow of the edge, as well as pointers to the head and tail nodes.

Due to the simple nature of our requirements, we did not use any libraries such as the C++ Standard Template Library (STL) [PSLM00] or the Library of Efficient Data types and Algorithms (LEDA) [MN00] to implement the structures. Even though LEDA includes types and algorithms for network applications, we nonetheless chose to avoid it due in part to a perception that more general libraries might result in a performance

penalty. A newer library that shows promise is the Generic Graph Component Library (GGCL) of Lee, Siek, and Lumsdaine [LSL99], which was developed after our work began. The authors of GGCL said that it is five to seven times faster than LEDA, validating our concerns about the performance of LEDA.

One pleasant side effect of our decision to avoid third-party libraries is that the code is comparatively portable because it does not depend on such libraries or "newer" features in C++, like multiple inheritance and templates.

The pseudoflow solver is also represented as a class. This allows us to create a subclass to override the behavior of the base solver. This is how the simplex solver is implemented with minimal disruption to the highest and lowest label solvers. The entire implementation of the pseudoflow algorithm is a single program. The various combinations of heuristics are selected via command line options.

The implementation makes extensive use of small, inline functions. Through the use of conditional compilation in the C preprocessor, these can be compiled inline for performance or out-of-line for debugging and profiling. This allows the use of a profiler such as `gprof` [GKM82] to collect very detailed information about the inner workings of the implementation without sacrificing performance during normal operations.

The initial development was conducted on a Pentium-based system running FreeBSD [Leh96]. Subsequent development was done on a laptop computer running Windows 98 using the Cygwin tools [Noe98]. Performance testing was done on Sun Microsystems computers running Solaris. On all systems, we used the GNU compiler tools, including versions 2.7.2.1, 2.8.1, 2.91.57, and 2.95.2. The code was compiled using the "-O2" optimization flag.[5] In addition, the source code was written using the `noweb` [Ram94] literate programming [Knu92] tool. The resulting documented code is over 230 pages after formatting with LaTeX. There are about 5,500 lines of embedded C++ code.

3.2.2 Push-relabel and Dinic's Implementations

The implementations of push-relabel and Dinic's algorithm we tested were written by Goldberg and Cherkassky [GC97]. Collectively these are known as PRF. These include the

[5] We experimented briefly with the -O4 flag and found that it did not improve the performance. We opted for -O2 to reduce the possibility of optimizer-induced errors in the executable programs.

highest-label and FIFO variants of the push-relabel algorithm. The push-relabel variants use both gap and global relabeling. These were selected as the baselines both because they were readily available, and because Goldberg and Cherkassky indicate that they are faster than other recent implementations such as Anderson and Setubal [AS93]. We considered the push-relabel implementation of Badics [BB93], but found memory management bugs [And97], and so eliminated it from further consideration.

Each variant of push-relabel is implemented as a stand-alone program rather than selecting the behavior from the command line. The programs are written in C and use no external libraries. They were compiled with the GNU tools under FreeBSD, Solaris and Cygwin. The "-O4" optimization flag was used to compile the PRF implementations since that was the optimization flag used by Goldberg and Cherkassky [GC97].

The PRF implementation of Dinic's algorithm was also used, again because it was easily available and Goldberg and Cherkassky [GC97] indicate that it is significantly faster than other available implementations.

The PRF code puts a premium on performance and has no additional documentation. In contrast to our fine-grained implementation of the pseudoflow algorithm, the PRF implementations feature large, monolithic functions that maximize performance, but make it difficult to profile the code with much precision.

3.2.3 Test Harness

Testing was conducted using an improved version of our workbench software [And97] written in Python [Lut96]. The workbench provides an object-oriented framework for manipulating the solvers, problem instances, and solutions. This provides a simple, intuitive interface that masks the differences between various solver and problem generators. This is achieved via a uniform *facade pattern* [GHJV94].

The result is much clearer and more intuitive than using the underlying programs directly. Consider using the `washington` generator (described later) to create an instance of the *rlg-long* graph[6] with 2^{14} nodes with a random number generator seed of 5. This uses function code 6, $dim1 = 64$, and $dim2 = 256$. The command would be:

```
washington -seed 5 6 64 256 rlg-long.max
```

[6]This type of graph and the parameters that define it are described in Section 3.4.2.

Note that it is difficult to recognize which number on the command line controls which parameter of the program.[7] These conventions vary from program to program, which further complicates the problem.

In our framework, this would be done as follows:

```
gen = RLGLong()
gen.seed = 5
gen.xParam = 14
gen.createFile('rlg-long.max')
```

To be sure, this is more verbose, but it is also fairly obvious what the various parameters do.

Each solver and generator has its own conventions for being invoked. For example, the PRF solvers read problem input on standard input and write solutions to standard output, whereas the pseudoflow solvers read and write files. The workbench provides a uniform interface regardless of the underlying programs.

To invoke a PRF solver (e.g., `h_prf`) and the pseudoflow solver (`pfs`) from the command line, one would use:

```
h_prf < test.max > test.flow
pfs test.max test.flow
```

Note that the PRF solver uses the shell I/O redirection operators, "<" and ">", whereas the pseudoflow solver does not. In the workbench, we use the `solve` function on a solver to invoke the underlying program, thus concealing the differences in how the programs handle input and output:

```
inst = ProbInstance.getInstance('test.max')
hprf = HighestLabel()
soln1 = hprf.solve(inst)
pfs = LowestLabel()
soln2 = pfs.solve(inst)
```

The workbench also provides facilities to validate the solutions from solvers. At the very least, we check the value of the maximum flow against the value returned by the highest-label version of push-relabel. If we have reason to doubt a solution, especially

[7] The random number seed, 5, is comparatively intuitive, but we added that parameter with the "-seed" flag to call it out. It was not part of the original code.

the "benchmark," push-relabel highest-label solution, we can also perform more thorough checking by verifying the flow balance and capacity constraints and by identifying the source and sink sets of the minimum cut and comparing the flow value to the capacity out of the source set or into the sink set.

The CATS [GM99] maximum flow distribution provides a series of shell scripts to execute the solvers and generators. Our framework is much more sophisticated and portable than the shell scripts. The CATS scripts force all programs to conform to a single calling convention; all programs must read from standard input and write to standard output. This has led to minor modifications of the DIMACS instance generators to fit within this framework. The CATS scripts provide no error checking. Even the exit values of the programs are not checked.

On the other hand, the reusable solver framework of Ruark [Rua98] is much more sophisticated than our software. It relies heavily on the Microsoft COM [Rog97] system to integrate various tools. This allows solvers to be used from programs such as Excel. However, this essentially requires that the tools be used under Microsoft Windows because COM is not widely used or available on other operating systems. In contrast, our tools are written portably in Python, and they run on many platforms including Windows, FreeBSD, and Solaris. However, because Python on Windows includes extensions to interface with COM, we could make our framework available via COM as an option on Windows without requiring that the software run exclusively on Windows.

3.3 Hardware

As mentioned above, development and preliminary performance testing was conducted on a variety of systems. The results of the final performance tests presented here were conducted on a Sun Microsystems E420R system. The system had four SPARC V9 CPUs running at 450 MHz and a total of 2 GB of RAM. Although this was a multi-CPU system, we only ran one test at a time to avoid performance anomalies such as bus contention and cache conflicts.

We performed the machine calibration experiment, as suggested by the DIMACS Challenge Core Experiments [Dim90]. Table 3.1 shows the running times (in seconds) for the

two tests with and without compiler optimizations.

Optimization	Test 1			Test 2		
Level	real	user	system	real	user	system
no optimization	0.2	0.2	0.0	1.7	1.7	0.0
-O flag	0.1	0.1	0.0	0.8	0.8	0.0

Table 3.1: Average running times for DIMACS machine calibration tests on Sun E450 system.

The original intent of this test during the DIMACS Challenge was to prepare a "comparative study of the diverse programming environments used by Challenge participants and of sources of variation in runtime experiments" [Dim90]. Currently, it is a simple measure of hardware performance that can be used to help put absolute running times reported by various authors into some kind of relative perspective.

3.4 Methodology

3.4.1 Testing Procedures

Our testing methodology is based on the suggestions from DIMACS Core Experiments document [Dim90]. For each random problem type and particular size, we generated five specific instances, each with a different random seed. For each specific instance, we averaged the times over five runs. For nonrandom generators and data files, we averaged the times of five runs on a single instance. These 5 or 25 runs are considered one test point.

Although the testing was performed on a system that was basically idle, we chose to run the solvers on each specific instance five times to eliminate any variation due to other processes (e.g., system processes) running on the computer or due to other system-level issues that we did not have control over. The choice of five different random seeds is based on the methodology of Goldberg and Cherkassky [GC97].

We performed initial testing for all problem classes with modest-sized instances. During this phase we tried all possible combinations of heuristics for the pseudoflow algorithm. Based on these results, we identified a set of heuristics that was the best for each problem class for this particular instance size. Note that this combination of heuristics need not

be the best for all instance sizes within the problem class, and each problem class has its own set of such heuristics.

Since it would be unrealistic to expect most users to permute all of the possible combinations of pseudoflow heuristics to find the best combination for their particular problem(s), we chose two standard sets of heuristics. We used these three sets of heuristics (two standard and one specific to the instance) to perform a more thorough analysis.

After the initial testing, we examined the performance of the solvers on each problem class more closely. We generated a series of instances, each with roughly twice as many nodes as the previous instance. For each instance size within a class, we generated five instances of the random graphs. We ran the PRF solvers and the three heuristics for the pseudoflow solver against the various problem instances. This demonstrates how well the performance of the solvers scales with instance size.

All timings were done with the POSIX `times` system call [Pos88], and we only report the user CPU time. Execution times exclude the time taken to read the problem instance. In the case of the pseudoflow implementations, however, we do include the time to establish the initial normalized forest and the initial node labels.

3.4.2 Problem Instances

Most of our problem instances are created by the well-known generators used in the DIMACS Challenge. Many of these are also part of the CATS [GM99] maximum flow problems. In addition, we use some synthetic data from the open-pit mining problem as well as an actual data instance from a real mine. Finally, we have some minimum cut instances from scheduling problems. All of these are described below.

Except where otherwise noted, the generating programs are available from DIMACS and CATS, and the graphs are randomly generated based on a seed value.

3.4.2.1 Genrmf

The genrmf generator is from Goldfarb and Grigoriadis [GG88] and is also known as RMFGEN, especially in older literature. The dimensions of the graph are controlled by two parameters, a and b. The generated graph contains b frames, each composed of a^2 nodes. The frames are arranged in a square grid, a nodes on a side. Each node in a frame

has forward arcs to up to four immediate neighbors within the frame. Neighboring frames are connected with a^2 arcs that form a randomly-generated perfect matching between the nodes in two adjacent frames. The capacities of the arcs are controlled by two parameters: c_1 and c_2. Arcs within a frame have a fixed capacity of $c_2 a^2$. The capacities of arcs between frames are chosen uniformly from $[c_1, c_2]$. The values specified by DIMACS are $c_1 = 1$ and $c_2 = 10^4$.

Within this class, there are two subclasses, genrmf-wide and genrmf-long, that are controlled by a single parameter x. The number of nodes in the graph is 2^x. For genrmf-long graphs, $a = 2^{x/4}$ and $b = 2^{x/2}$. For genrmf-wide, $a = 2^{2x/5}$ and $b = 2^{x/5}$.

3.4.2.2 Washington

The washington generator was developed by Anderson and several seminar students at the University of Washington [GC97]. It can generate many different types of graphs. These types are sometimes referred to in the literature by a *function code* number, which is a parameter on the command line that determines the type of generated graph. The program takes five parameters: the function code, two dimensions ($dim1$ and $dim2$), a range (*range*), and the random seed.

Random Level The first graph type is the random level graph (rlg), which is generated via function code 2. The graph is a rectangular grid of $dim1$ rows and $dim2$ columns. Each node is connected to three nodes chosen at random in the next column. This is an acyclic graph. The capacities of the arcs between columns are uniform random numbers between zero and *range*, which is 10^4 as per the DIMACS suggestions. Like genrmf, there are two subtypes, rlg-long and rlg-wide, that are controlled by a single parameter x, which again specifies the total number of nodes in the graph, $n = 2^x$. For rlg-long, $dim1 = 64$ and $dim2 = 2^{x-6}$. For rlg-wide, $dim1 = 2^{x-6}$ and $dim2 = 64$. In other words, long graphs are 64 nodes wide and the depth varies, whereas wide graphs are 64 nodes deep and the width varies.

Line Moderate The second graph type is the line-moderate graph, which corresponds to function code 6. To generate a graph with 2^x nodes, we set $dim1 = 2^{x-2}$ and $dim2 = 4$. This will create a graph with four ($dim2$) nodes adjacent to the source

and four others adjacent to the sink. The *range* parameter specifies the out-degree of each node, and it is set to $2^{(x/2)-2} = \sqrt{n}/4$. Each vertex i (other than the source and sink) has up to $\sqrt{n}/4$ arcs to vertices chosen randomly from $[i+1, i+\sqrt{n}]$.[8] The capacities of these arcs are selected uniformly from $[1, 10^6]$. This graph is also acyclic.

Cheriyan The final graph type used from the washington generator was "designed by Cheriyan, and considered as a hard case for Goldberg's algorithm" [BB93]. It corresponds to function code 11. It is a deterministic graph that is not part of the recommended set of generators from CATS. We chose the parameters based on the values used by Badic and Boros [BB93]. To generate a graph with 2^x nodes, we set $dim1 = 1000$, $dim2 = (2^x - 2007)/40$, and $range = 10$.

We modified the washington program to accept a seed value for the random number generator on the command line. This allows us to consistently regenerate graphs based on the seed and the rest of the parameters.

3.4.2.3 Acyclic Dense

The acyclic-dense generator, AC, was originally developed by Waissi in Pascal and translated to C by Setubal [GC97]. For each node numbered k, there are arcs to every other node with a higher number—i.e. $k+1, k+2, \ldots, n$. In particular, the source has a link to every node in the graph, including the sink, and every node has an arc to the sink. The capacities of these arcs are selected randomly from $U[1, 10^6]$.

3.4.2.4 AK

The AK generator was developed by Goldberg and Cherkassky [GC97]. It creates instances which are hard for push-relabel. It is not available from the DIMACS site, but is distributed with the PRF solver implementations. It is also part of the CATS maximum flow package. This class of graphs is not random. For a specified size parameter, k, it generates a deterministic graph of approximately $4k$ nodes.

[8] A random number r is chosen from $U[i+1, i+\sqrt{n}]$. If $r > n$, then no arc is generated.

The graph contains two subgraphs, one which is designed to be hard for the highest-label and FIFO variants, and the other is designed to be hard for the wave variant. The graph is described in detail in the appendix of Goldberg and Cherkassky [GC97].

3.4.2.5 Synthetic Mining Data

We use synthetic data from Hochbaum and Chen [HC00] that simulates data for the open-pit mining problem (e.g., see [Joh68]). The generator produces files of various sizes up to 54,000 blocks/nodes. The successor blocks above each block follow specific patterns used in the mining industry and are described by Hochbaum and Chen. With 54,000 blocks, this results in 410,330 arcs. These instances are quickly solved by the implementations we are testing, so we concatenated the instances together to make larger instances, four times the original size—i.e., 216,000 blocks and 1,657,580 arcs. We tested with three separate instances of these files, which are called *mine1*, *mine2*, and *mine3*.

3.4.2.6 Real Mining Data

We obtained a data file from a nickel laterite mine in Australia from a company that wishes to remain anonymous. The data file consists of 162,302 blocks/nodes and 942,144 arcs (the successors of each block are the closest five blocks in the layer above it). We only have a single instance of such data, which is called *Pit2*.

3.4.2.7 Scheduling Data

Möhring et al. [MSSU03] model project scheduling problems with start-time dependent costs as minimum cut problems. The authors also describe an algorithm for solving resource-constrained project scheduling problems using Lagrangian relaxation to handle the resource constraints. Each relaxation results in a minimum cut problem on a graph.

The scheduling problems consist of J jobs, which are scheduled non-preemptively over a time period T. Let S_j be the start time for each job. Every job j incurs a cost of w_{jt} if j begins at time t (i.e., if $S_j = t$). Let d_{ij} be the integral length of time between two jobs i and j. For any feasible schedule, $S_j \geq S_i + d_{ij}$, indicating that j must start at least d_{ij} time units after i. The objective is to minimize the sum of the costs w_{jt} for all jobs subject to the temporal constraints and the constraint that each job is run exactly once.

One of the main results of Möhring et al. [MSSU03] is that this scheduling problem can be modeled as a minimum cut problem. There is a vertex v_{jt} for each job and time period combination, and there is a source and a sink node. There are two main types of arcs in the graph: *assignment arcs* (v_{jt}, v_{jt+1}) and *temporal arcs* $(v_{it}, v_{jt+d_{ij}})$. There are also arcs from the source to all nodes v_{j0} and from nodes v_{jT} to the sink. The capacities of the assignment arcs are $c(v_{jt}, v_{jt+1}) = w_{jt}$, and the temporal arcs have infinite capacity. Möhring et al. proved that the value of an optimal solution equals the capacity of any minimum cut in the graph.

Möhring et al. [MSSU03] also consider resource-constrained scheduling problems. They use Lagrangian relaxation to vary the costs w_{jt} and solve for a minimum cut in the graph with updated capacities. The authors currently solve these minimum cut problems with the push-relabel algorithm.

The authors provided us with a number of scheduling instances. We selected three instances that were considered to be the most difficult to solve. Most of the individual minimum cut instances are quickly solved in less than a second. Therefore, we report the time to solve an entire sequence of 51 minimum cut problems. The names and sizes of the instances are shown below in Table 3.2.

Name	Nodes	Arcs
j6	11,617	27,487
j16	18,393	44,416
j56	20,565	60,567

Table 3.2: Sizes of three most difficult scheduling problem instances.

3.5 Overall, Best-Case Performance

3.5.1 Initial Summary

Table 3.3 below is a summary of the performance of the pseudoflow solver versus the PRF implementations for the modest-sized instances. For these instances, we permuted all possible heuristic combinations for the pseudoflow algorithm, as described in Section 3.4.1.

Each row in Table 3.3 shows the execution times of the various solvers for a given problem class. Each cell contains the execution time in seconds and the relative performance

of each solver as a percentage of the execution time of the best solver for that problem class. So, the best solver has a relative time of 100, and all others are 100 or more. The best execution time for a given instance for all solvers is highlighted in bold font.

The solvers used throughout this chapter are:

Dinic: the PRF implementation of Dinic's algorithm.

FIFO: the FIFO push-relabel algorithm.

HL: the highest-label push-relabel algorithm.

pfs-C: the first standard pseudoflow solver using the highest-label algorithm with simple initialization, constant initial distance labels, wave branch management, pre-order branch searching, and global relabeling with a frequency of $4n$, that is simple-const-highest-wave-pre-4.

pfs-D: the second standard pseudoflow solver, which also uses the highest label algorithm and simple initialization, but with distance-to-deficit initial node labels, lifo branch management, pre-order branch searches, and no global relabeling—i.e., simple-deficit-highest-lifo-pre-0.

pfs-EB: the combination of pseudoflow heuristics that we found ran best for the modest-sized instance of this class. This combination is unique for each problem class.

In Table 3.3, we can see that pfs-EB has the best performance in 7 out of 11 cases. Furthermore, pfs-EB is within 50% of the best push-relabel for all cases except rlg-long.

We can also see that one or the other of our standard heuristic combinations is within 45% of the best push-relabel in 7 out of 11 cases. Also, all three of the pseudoflow heuristic combinations beat the Dinic solver in all cases except cheriyan.

From these modest-sized instances, it is clear that the performance of the new pseudoflow solver is certainly competitive with the mature push-relablel solvers, and the pseudoflow solver is generally superior to the Dinic solver.

3.5.2 Variations within Initial Summary Data

For the randomly generated problem classes, we generated five specific instances for each problem problem size. The data in Table 3.3 represent the average times for each solver over all five specifc instances. In Table 3.4 we look at the variations in the run times for specific instances within the same random problem class for each solver.

	PRF Solvers			Pseudoflow Solvers		
Problem Class	Dinic	FIFO	HL	pfs-C	pfs-D	pfs-EB
ac.2048	12.343	6.200	3.888	2.054	3.011	**1.752**
n =2,048 m =2,096,128	705	354	222	117	172	100
ak.8000	120.730	17.332	10.414	5.276	83.152	**0.036**
n =32,006 m =48,007	335,361	48,144	28,928	14,656	230,978	100
cher-wide.16	4.682	16.409	53.196	95.986	320.072	**2.554**
n =65,527 m =72,875	183	642	2,083	3,758	12,532	100
genrmf-long.16	64.995	2.403	**0.692**	1.527	0.859	0.788
n =15,488 m =71,682	9,392	347	100	227	124	114
genrmf-wide.16	60.09	10.881	6.831	14.844	26.854	**3.688**
n =16,807 m =80,262	1,629	295	185	402	728	100
line-mod.16	16.831	2.503	**2.158**	9.694	2.860	2.829
n =16,386 m =522,249	780	116	100	449	133	131
rlg-long.16	37.474	1.284	**0.436**	1.454	1.804	1.363
n =65,538 m =196,544	8,595	294	100	333	414	313
rlg-wide.16	11.344	2.245	**1.375**	1.995	3.086	1.921
n =65,538 m =195,584	825	163	100	145	224	140
sched 6, 16, 56	164.102	33.976	66.434	64.53	132.024	**27.288**
(total time, all instances)	601	125	243	256	484	100
mine1-3	158.23	35.43	37.65	20.79	21.87	**17.56**
n =216,000 m =1,657,580	901	202	214	118	125	100
Pit2	11.53	8.45	8.47	8.88	5.13	**2.03**
n =162,302 m =942,144	568	416	417	438	253	100

Table 3.3: Overall summary for modest-sized instances. In each cell, the average solving time in seconds is on top. The best solver time across all solvers is in bold font. The solving time relative to the best is on the bottom, where the best solver for the problem class has a relative time of 100.

Each cell in Table 3.4 describes the variation in running time for a specific solver/problem class pair. The minimum and maximum running times are expressed as a percentage deviation from the average reported in Table 3.3 (the minimum is represented as a negative number and located above the maximum in the cell). For each minimum and maximum, we also note which instance (0 through 4) in the set of specific instances produced that time. For each solver, the smallest minimum and largest maximum are set in bold font.

	PRF Solvers						Pseudoflow Solvers					
Problem Class	Dinic		FIFO		HL		pfs-C		pfs-D		pfs-EB	
genrmf-long.16	-13.56%	(1)	-5.38%	(4)	-10.06%	(1)	-5.47%	(3)	-14.81%	(4)	-13.96%	(4)
	7.31%	(0)	11.10%	(2)	**30.13%**	(2)	8.65%	(2)	24.51%	(2)	21.07%	(2)
genrmf-wide.16	-20.56%	(1)	-14.37%	(0)	-14.63%	(0)	-7.79%	(2)	-3.11%	(2)	-11.40%	(0)
	23.73%	(3)	9.69%	(1)	14.39%	(3)	8.46%	(1)	5.27%	(1)	7.31%	(3)
line-mod.16	-5.40%	(0)	**-27.20%**	(1)	-27.43%	(1)	-18.38%	(4)	-12.30%	(1)	-12.83%	(1)
	5.57%	(3)	7.24%	(0)	7.23%	(4)	28.70%	(1)	19.25%	(4)	19.06%	(4)
rlg-long.16	**-23.41%**	(2)	-23.08%	(2)	**-31.19%**	(0)	**-34.65%**	(1)	-51.66%	(0)	**-36.91%**	(0)
	22.18%	(3)	**39.05%**	(3)	28.44%	(4)	**111.47%**	(0)	63.97%	(3)	**82.07%**	(2)
rlg-wide.16	-12.94%	(2)	-10.21%	(1)	-30.03%	(1)	-31.82%	(4)	**-57.17%**	(1)	-20.66%	(1)
	10.75%	(0)	12.06%	(4)	22.78%	(4)	44.48%	(1)	**99.97%**	(4)	31.92%	(4)

Table 3.4: Performance variations within instances of random classes. Each cell shows the minimum (top) and maximum (bottom) run times relative to the average for the five instances tested. The bold entries in each column indicate the most extreme minimum and maximum for the solver across all of the instance classes.

Within a problem class, there is a considerable of variation in the running times for the various solvers. Sometimes the variation for a solver within a given problem class is comparatively small—e.g., Dinic on the line-moderate and pfs-D on genrmf-wide are approximately ±5%. However, there are also fairly large variations too—e.g., pfs-D on rlg-wide varies by approximately a factor of four (the minimum is 50% below the average, and the maximum is 100% above the average.)

Within any given problem class, the variation does not seem attributable to any one instance. The instance that was the fastest or slowest for one solver is not necessarily the same instance that was fastest or slowest for another solver. In fact, when looking at the line-moderate and rlg-wide classes and the pfs-C and pfs-D solvers, the fastest for one solver is the slowest for the other solver and vice versa. For example, the fastest instance for pfs-C on the line-moderate data was number 4, but that instance was the slowest for pfs-D.

With regard to the solvers, the pseudoflow solvers seem to be less stable than either the

Dinic or push-relabel solvers. The pseudoflow solvers created the most extreme minima and maxima. In some cases (e.g., pfs-C on rlg-long, pfs-D on rlg-wide, and pfs-EB on rlg-long), the smallest minimum and the largest maximum for the solver both occur within the same problem class. This suggests one or more outliers in that problem class that are distorting the average, which would affect both the maximum and minimum since they are expressed relative to the average.

Regardless of the details of the causes for the extremes, the performance of the pseudoflow solvers appears to be more sensitive to the input data than either the push-relabel or Dinic solvers.

3.5.3 AC

Performance for the AC family of graphs is shown in Figure 3.1. The pfs-EB heuristic combination is simple-const-lowest-wave-pre-0. Note that data is missing for HL and FIFO for the case where $n = 4096$. This is due to a bug in these solvers that causes them to return an incorrect result.[9]

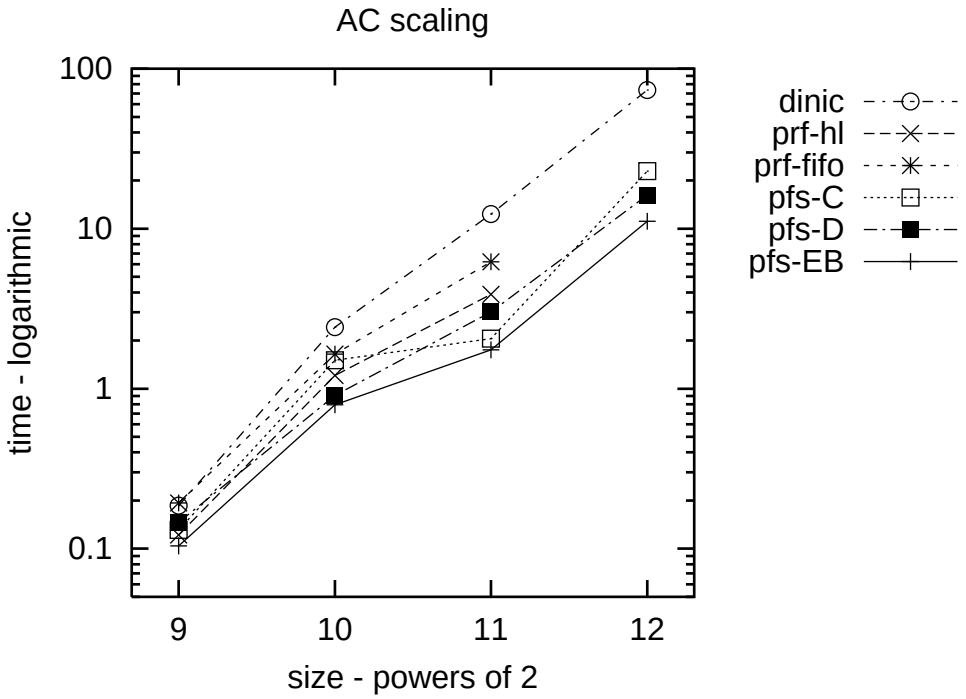
All of the pseudoflow solvers beat the FIFO and Dinic solvers. The pfs-EB solver beats the highest-label push-relabel solver for all cases. All of the pseudoflow solvers beat the highest label solver for $n = 2048$.

3.5.4 AK

Performance for the AK family of graphs is shown in Figure 3.2. The pfs-EB combination of heuristics is saturate-const-highest-lifo-pre-0, although all of the heuristic combinations that use the saturate-all initialization method solve the ak.8000 instance in 0.1 seconds or less.

The pfs-EB heuristic (with saturate-all) is two to three orders of magnitude faster than the highest-label push-relabel. The pfs-C solver is typically faster than push-relabel, but it runs slightly slower than the highest-label algorithm for $n = 64,006$ nodes. The pfs-D solver is consistently poor, although it beats the Dinic solver.

[9]Because we performed our testing on a larger and faster system, we were able to test larger instances than Goldberg and Cherkassky did. For example, they do not report results for the AC graph with 4,096 nodes.



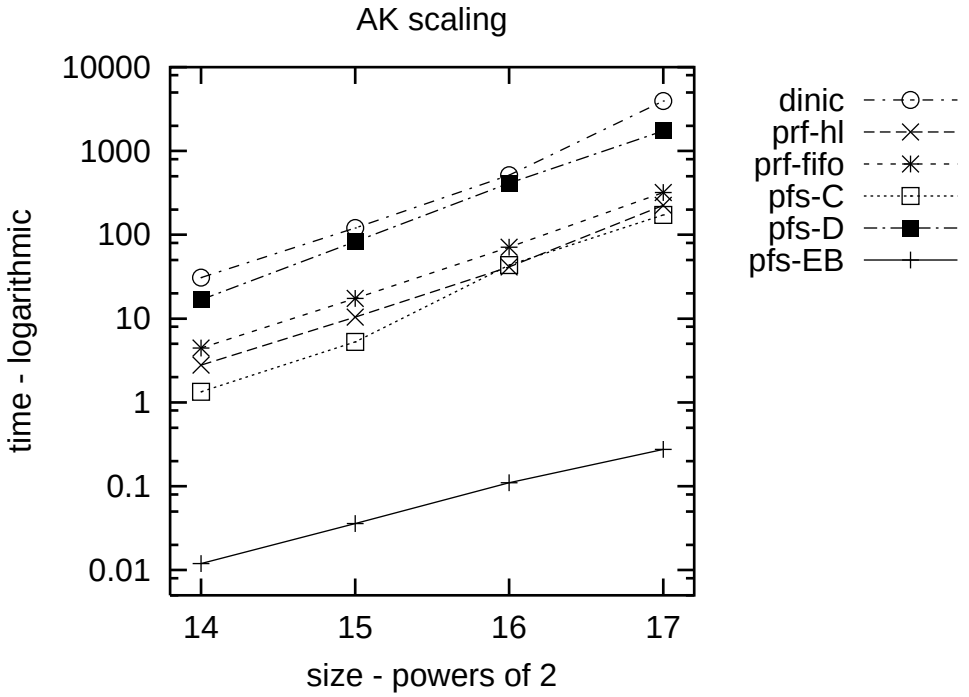
Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
512	130,816	0.186	0.122	0.193	0.131	0.146	0.104
1024	523,776	2.425	1.209	1.645	0.616	1.503	0.793
2048	2,096,128	12.343	3.888	6.200	1.786	2.054	1.752
4096	8,386,560	73.672	*error*	*error*	22.925	16.202	11.125

Figure 3.1: Performance for AC family graphs. pfs-EB is simple-const-lowest-wave-pre-0.

The saturate-all initialization method excels for the AK family because of the structure of the graph. For a size parameter k, the graph contains four paths of length k with deterministic capacities based on k. By construction, along two of these paths the capacities of each node are balanced—the in-capacity matches the out-capacity, and the optimal solution involves having all arcs saturated. The in- and out-capacities of the nodes along the second pair of paths are balanced, except that there is an arc of unit capacity between a node in the first path and a node in the second path.

After saturating the arcs in the graph, all of the nodes in the first two paths have zero deficit. In the other two paths, there are k nodes with one unit of excess and k nodes with one unit of deficit. The pseudoflow solver performs k mergers to return the single unit of

excess from each of the k excess nodes to the k deficit nodes, at which point the solution is optimal.



Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
16,006	24,007	30.776	2.796	4.458	1.342	16.802	0.012
32,006	48,007	120.730	10.414	17.414	5.276	83.152	0.036
64,006	96,007	513.598	41.266	71.174	43.496	412.374	0.110
128,006	192,007	3,945.000	224.936	318.516	172.808	1,729.816	0.276

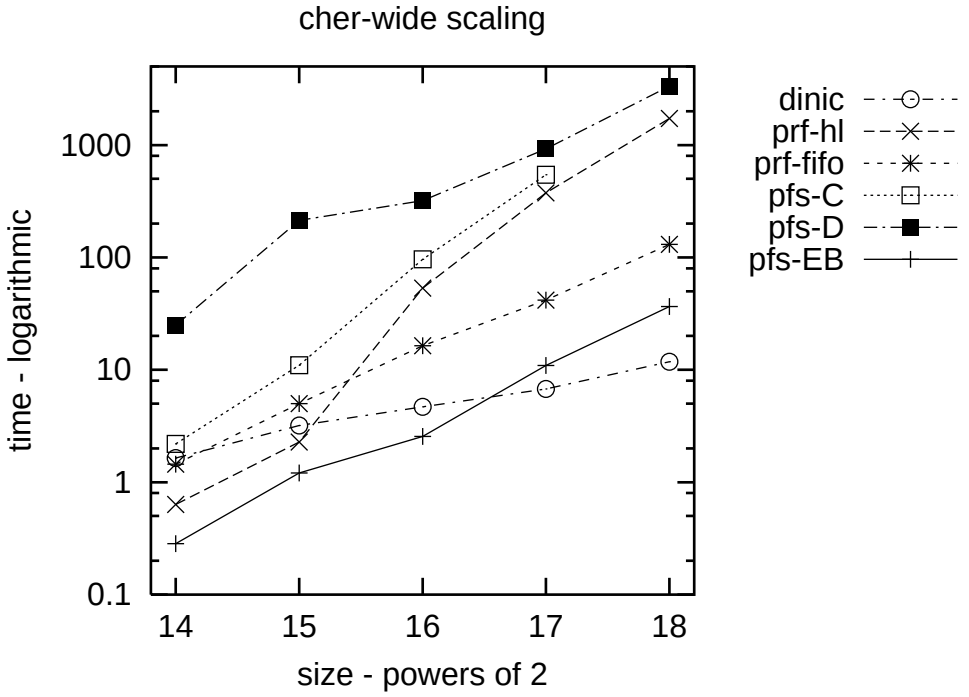
Figure 3.2: Performance for AK family graphs. pfs-EB is saturate-const-highest-lifo-pre-0.

3.5.5 Cheriyan

Performance for cheriyan is shown in Figure 3.3. The pfs-EB heuristic combination is shortest-const-highest-lifo-pre-1.

The results are not very robust. The Dinic solver performs relatively poorly with smaller instances, but it scales very well. On the two largest instances, it is the fastest of all of the solvers. The pfs-EB and FIFO push-relabel solvers scale fairly consistently. On the largest instances, they both beat the highest-label push-relabel, which works well with smaller instances, but does very poorly on instances with 2^{16} or more nodes. The

pfs-D solver consistently performs the worst.

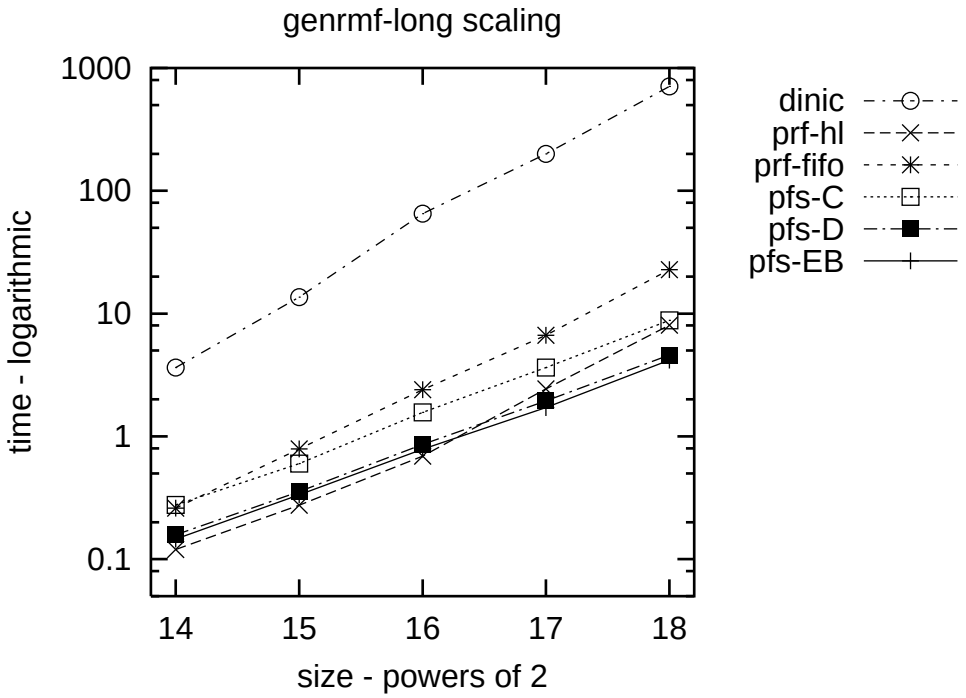


Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
16,367	18,799	1.640	0.634	1.446	2.188	24.884	0.284
32,767	36,839	3.194	2.282	5.000	10.982	213.142	1.208
65,527	72,875	4.682	53.196	16.409	95.986	320.072	2.554
131,047	144,947	6.744	374.078	41.644	543.194	925.816	10.942
262,127	289,135	11.784	1721.740	130.482	*error*	3335.040	36.660

Figure 3.3: Performance for cheriyan family graphs. pfs-EB is shortest-const-highest-lifo-pre-1.

3.5.6 Genrmf Long

Performance for the genrmf-long family of graphs is shown in Figure 3.4. The pfs-EB heuristic combination is greedy-deficit-highest-lifo-pre-0. The highest-label solver is the faster of the push-relabel solvers. Although the highest-label push-relabel solver is faster than the three pseudoflow solvers for problem instances up to $n = 2^{16}$, for larger instances the pfs-D and pfs-EB solvers are faster. Also, all of the other solvers beat the Dinic solver.

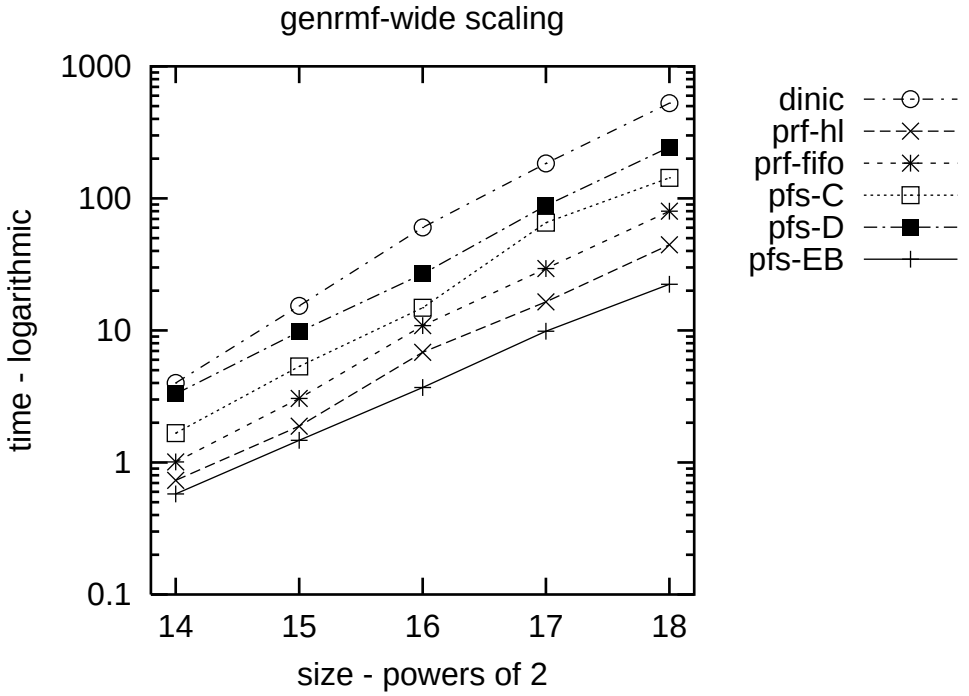


Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
15,488	71,687	3.636	0.120	0.260	0.276	0.158	0.146
30,589	143,364	13.609	0.275	0.790	0.601	0.356	0.334
65,536	310,040	64.995	0.692	2.403	1.572	0.859	0.788
130,682	625,537	199.318	2.439	6.674	3.639	1.941	1.721
270,484	1,306,607	706.585	8.058	22.742	8.807	4.608	4.185

Figure 3.4: Performance for genrmf-long family graphs. pfs-EB is greedy-deficit-highest-lifo-pre-0.

3.5.7 Genrmf Wide

Performance for the genrmf-wide family of graphs is shown in Figure 3.5. The pfs-EB heuristic combination is shortest-const-highest-wave-pre-0. The pfs-EB solver consistently beats the push-relabel solvers, but the standard heuristic combinations fail to match the performance of either of the push-relabel solvers. Again, the Dinic solver is the worst solver.



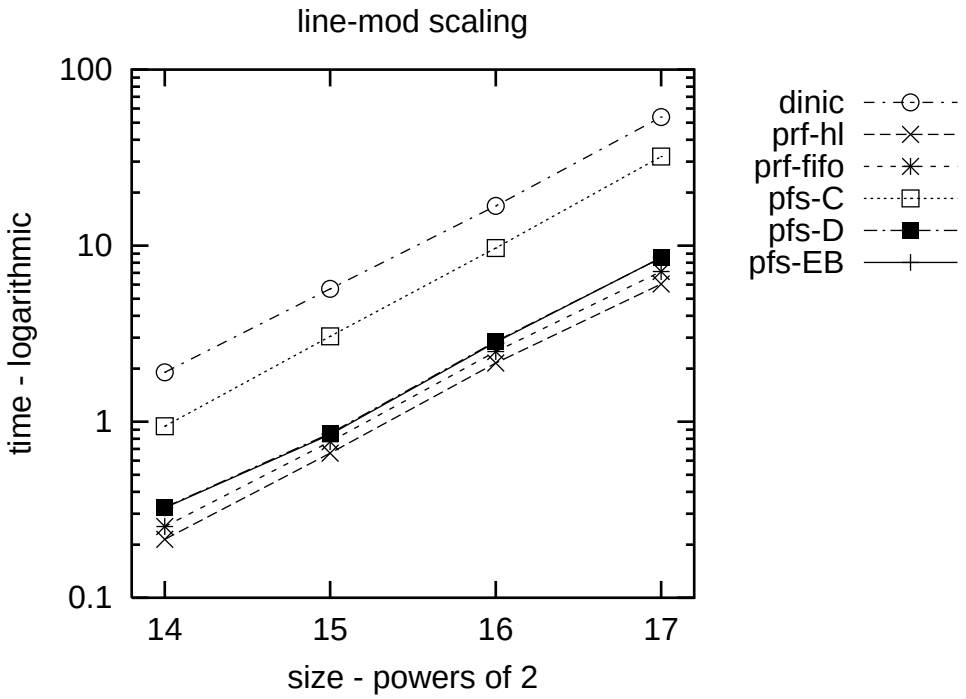
Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
16,807	80,262	4.008	0.732	1.009	1.670	3.298	0.578
32,768	157,696	15.316	1.885	3.059	5.326	9.764	1.468
63,503	307,44	60.090	6.831	10.881	14.844	3.688	4.429
135,531	660,45	183.938	16.477	29.409	65.393	88.241	9.863
259,308	1,267,875	525.328	44.593	80.110	143.049	244.186	22.373

Figure 3.5: Performance for genrmf-wide family graphs. pfs-EB is shortest-const-highest-wave-pre-0.

3.5.8 Line Moderate

Performance for the line-moderate family of graphs is shown in Figure 3.6. The pfs-EB heuristic combination is path-deficit-highest-wave-pre-0. Note that the number of arcs is approximate because the number of arcs generated is random, based on the random number seed.

Again, the Dinic solver is the worst followed by pfs-C. The other pseudoflow solvers are not as fast as the push-relabel solvers, but they are within 50% of both of the push-relabel solvers. The performance of the pfs-D and pfs-EB solvers is nearly identical, and hence they are hard to discern in the graph.

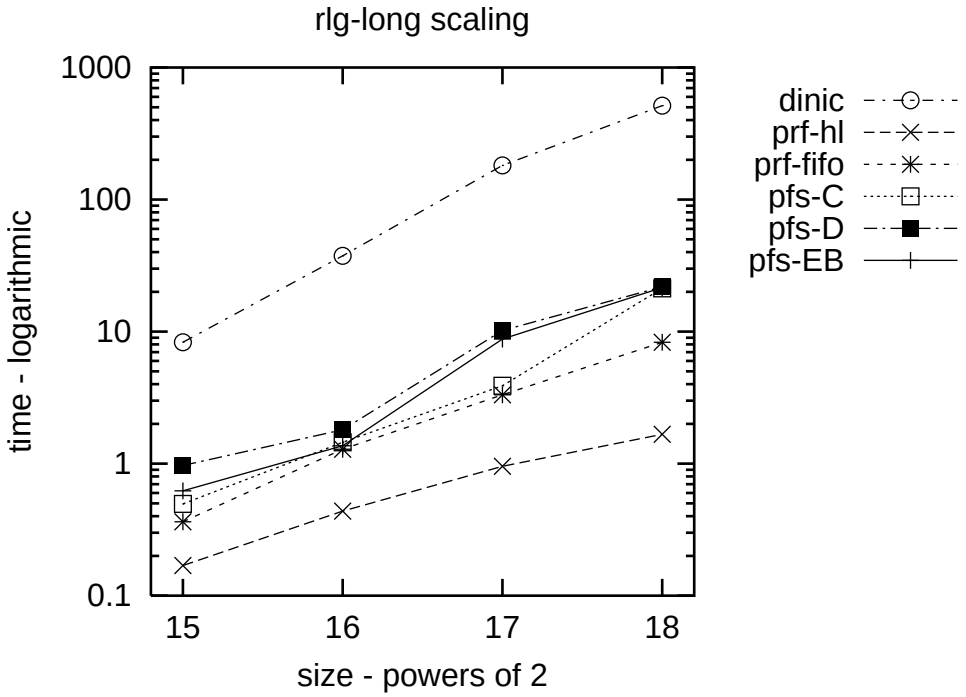


Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
16,386	522,249	1.910	0.215	0.254	0.944	0.326	0.324
32,770	1,470,473	5.679	0.662	0.771	3.050	0.858	0.846
65,538	4,186,093	16.831	2.158	2.503	9.694	2.860	2.829
131,074	11,910,889	53.776	6.060	7.120	32.075	8.531	8.557

Figure 3.6: Performance for line-moderate family graphs. pfs-EB is path-deficit-highest-wave-pre-0. The performance of the pfs-D and pfs-EB solvers are nearly identical and are therefore hard to differentiate in the chart.

3.5.9 RLG Long

Performance for the rlg-long family of graphs is shown in Figure 3.7. The pfs-EB heuristic combination is simple-deficit-highest-wave-pre-4. None of the pseudoflow solvers is as fast as either of the push-relabel solvers, but they are all faster than the Dinic solver. The performance of all three pseudoflow solvers is rather erratic; the plots all have "kinks," and they all converge at $n = 2^{18}$.



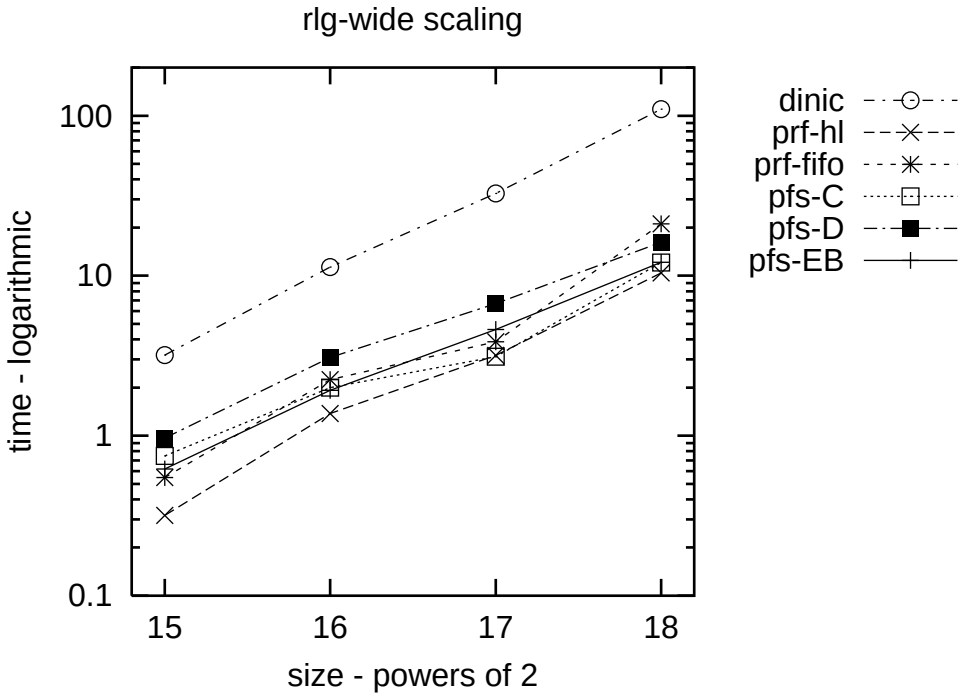
Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
32,770	98,240	8.299	0.169	0.363	0.495	0.969	0.621
65,538	196,544	37.474	0.436	1.284	1.454	1.804	1.363
131,074	393,152	181.188	0.956	3.320	3.892	10.177	8.792
262,146	786,368	514.118	1.671	8.305	21.224	21.841	21.534

Figure 3.7: Performance for rlg-long family graphs. pfs-EB is simple-deficit-highest-wave-pre-4.

3.5.10 RLG Wide

Performance for the rlg-wide family of graphs is shown in Figure 3.8. The pfs-EB heuristic combination is simple-const-highest-wave-pre-0.

The highest-label solver is the faster of the push-relabel solvers, and it is faster than all of the pseudoflow solvers. The FIFO push-relabel solver is faster than the pseudoflow solvers for smaller instances, but the performance of FIFO solver diverges at $n = 2^{18}$ where all the pseudoflow solvers beat the FIFO solver. Although pfs-C beats the highest-label push-relabel solver for $n = 2^{17}$, this appears to be an anomaly. Again, the Dinic solver is the worst.



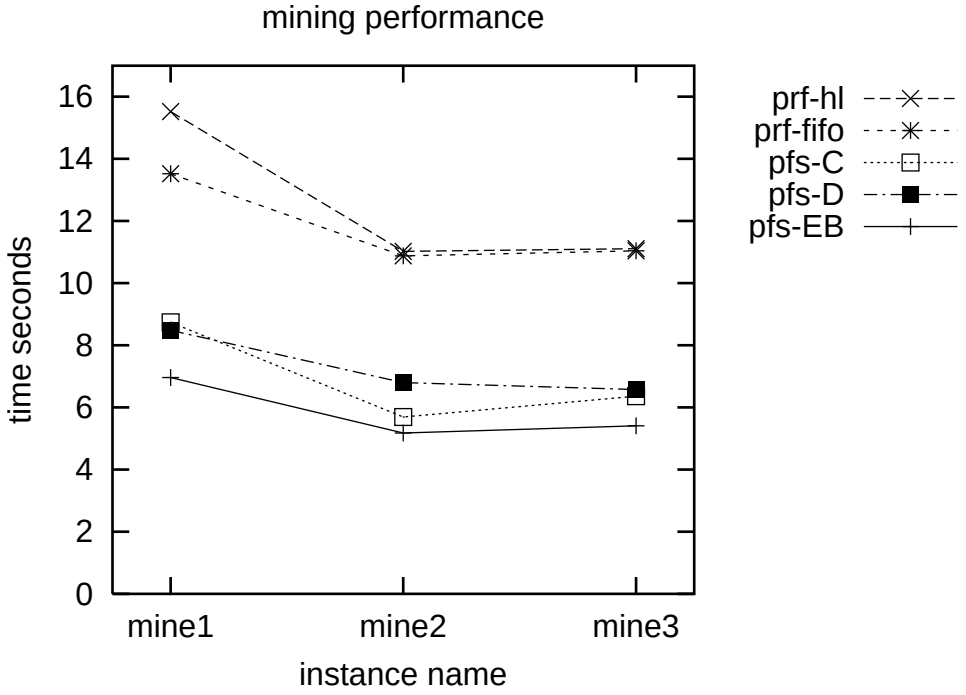
Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
32,770	97,792	3.192	0.318	0.547	0.748	0.968	0.620
65,538	195,584	11.344	1.375	2.245	1.995	3.086	1.921
131,074	391,168	32.679	3.179	3.866	3.122	6.691	4.619
262,146	782,336	110.066	10.405	21.094	12.090	16.206	12.150

Figure 3.8: Performance for rlg-wide family graphs. pfs-EB is simple-const-highest-wave-pre-0.

3.5.11 Mining Data

The performance for the synthetic mining data is shown in Figure 3.9 below. (The summary data in Table 3.3 was a sum of the times of the three instances.) The pfs-EB heuristic combination is simple-const-highest-wave-pre-0.

The pseudoflow solvers are consistently faster than the PRF solvers. Note that the FIFO push-relabel solver is faster than the highest-label push-relabel solver in all three cases.



Name	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
mine1	71.41	15.52	13.52	8.74	8.48	6.96
mine2	48.87	11.02	10.88	5.69	6.80	5.18
mine3	37.95	11.11	11.04	6.36	6.58	5.41

Figure 3.9: Times to solve three synthetic mining instances. pfs-EB is simple-const-highest-wave-pre-0. Dinic is not plotted in the figure due to the scale of the data.

In addition to the synthetic mining data, we tested with one data file from a real mine, *Pit2*. The performance for all of the solvers on this file is shown in Table 3.3. The pfs-EB heuristic combination is path-const-lowest-lifo-pre-0. The pfs-C pseudoflow solver

is slightly slower than the push-relabel solvers, but pfs-D and pfs-EB are considerably faster than the push-relabel solvers. The FIFO push-relabel solver is slightly faster than the highest-label push-relabel solver.

Although the synthetic instances and *Pit2* are all instances of the open-pit mining problem, the pfs-EB heuristic combinations are different. In particular, the highest-label pseudoflow algorithm was best for the synthetic instances, but the lowest-label pseudoflow algorithm was best for *Pit2*.

We suspected this was due to some details about the specific data within the data files. Hochbaum and Chen [HC00] computed the ratio of the average weights of the positive blocks in a mine divided by the average weights of the negative blocks. Let this ratio be R. For the synthetic instances, R is approximately 0.5, whereas $R = 15.6$ for *Pit2*.

We hypothesized that the highest-label algorithm works better for lower values of R, and the lowest-label algorithm works better for higher values of R. To test this, we took the *mine1* synthetic instance and perturbed the values of the positive blocks in the mine to create a series of instances with varying values for R. We solved each instance with both the highest and lowest label algorithms. The results are shown in Figure 3.10.

This confirms our hypothesis: for smaller values of R, the highest label algorithm is faster than the lowest label algorithm. For values of R greater than 10, the lowest label algorithm is slightly faster.

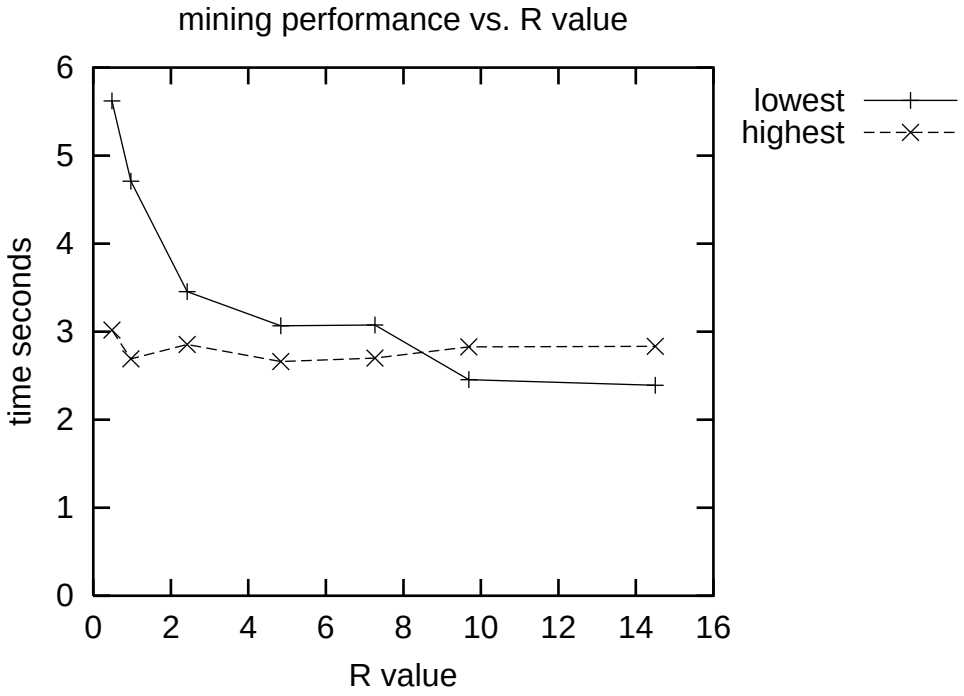
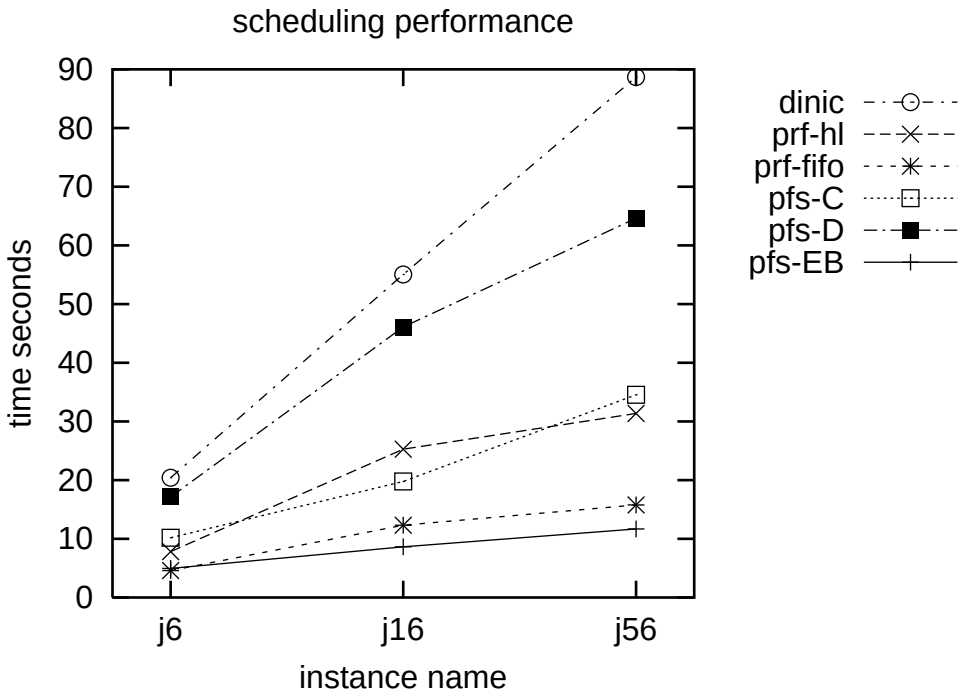


Figure 3.10: Solution times for the highest- and lowest-label pseudoflow algorithms on a series of instances derived from *mine1*. The times are plotted against the value of R for each instance.

3.5.12 Scheduling Data

The solution times for the three scheduling minimum cut instances (Table 3.2) are shown in Table 3.11 below. (The summary data in Table 3.3 was a sum of the times of these three instances.) The pfs-EB heuristic combination is simple-const-highest-lifo-pre-0.

The FIFO push-relabel solver is faster than highest-label push-relabel, and it is faster than the standard pseudoflow solvers. For the smallest instance, *j_6*, FIFO is slightly faster than pfs-EB, but for the other instances, pfs-EB is significantly faster.



Name	Nodes	Arcs	Dinic	HL	FIFO	pfs-C	pfs-D	pfs-EB
j_6	11,617	27,487	20.41	8.80	4.85	10.20	17.14	4.95
j_16	18,393	44,416	55.01	25.71	12.79	19.79	46.10	8.62
j_56	20,565	60,567	88.69	31.92	16.34	34.54	64.62	11.68

Figure 3.11: Times to solve three scheduling instances. pfs-EB is simple-const-highest-lifo-pre-0.

3.6 Study of the Performance of the Heuristics

The pfs-EB heuristic combination based on the performance for the modest-sized instance of each class is shown in Table 3.5. (The heuristic options are summarized in Table 2.1 in the previous chapter, and the instance sizes are shown in Table 3.3.)

Instance	Initial-ization	Initial Label	Root Label	Branch Management	Search Order	Relabel Frequency
AC	simple	const	lowest	wave	pre	0
AK	saturate	const	highest	lifo	pre	0
cheriyan	shortest	const	highest	lifo	pre	1
genrmf-long	greedy	deficit	highest	lifo	pre	0
genrmf-wide	shortest	const	highest	wave	pre	0
line-moderate	path	deficit	highest	wave	pre	0
rlg-long	simple	deficit	highest	wave	pre	4
rlg-wide	simple	const	highest	wave	pre	0
mine	simple	const	highest	wave	pre	0
Pit2	path	const	lowest	lifo	pre	0
sched	simple	const	highest	lifo	pre	0

Table 3.5: Best-case heuristics for pseudoflow solver on the modest-sized problems by instance type.

3.6.1 Initialization

We can see in Table 3.5 that the simple initialization scheme is the best in about half of the cases. Greedy and path initialization together account for the next largest portion. As we describe in Section 3.5.4, saturate-all is especially effective on AK graphs. However, it is not effective for any of the other problem cases.

The path and greedy initialization schemes both build branches and push flow into the network. The difference is that the path method stops as soon as it encounters a node with no single arc of sufficient capacity to handle the excess, whereas the greedy technique will push the flow along multiple arcs.

In some respects, these initialization schemes perform a simplified version of the main algorithm that merges from strong nodes to weak ones. Therefore, we might expect that these initialization schemes would reduce the amount of work done by the main algorithm at the expense of spending more time during initialization.

To investigate this hypothesis, we looked at the instances where the best heuristic combination used either path or greedy initialization—Pit2, line-moderate, and genrmf-long. We also included mine1 because we would expect that greedy and path should both perform well, even though simple initialization was slightly better during testing.

Table 3.6 shows the data for these instances using simple, greedy, and path initialization. We show the initialization time, solving time, the total time (initialization plus solving), and the number of mergers performed.

Instance	Initial Label	Initialization Method	Init Time	Solve Time	Total Time	Percent Init	Number of Mergers
Pit2	const	simple	0.33	1.98	2.31	14.29%	111,944
		greedy	0.59	1.47	2.06	28.64%	55,391
		path	0.59	1.46	2.05	28.78%	55,391
mine1	const	simple	0.39	6.41	6.80	5.74%	298,396
		greedy	0.81	6.17	6.98	11.60%	236,099
		path	0.83	6.21	7.04	11.79%	236,099
line-moderate	deficit	simple	1.43	1.59	3.02	47.35%	116,330
		greedy	2.36	1.72	4.08	57.84%	114,092
		path	1.41	1.55	2.96	47.64%	116,330
genrmf-long	deficit	simple	0.18	0.66	0.84	21.43%	138,692
		greedy	0.19	0.60	0.79	24.05%	136,960
		path	0.19	0.61	0.80	23.75%	133,727

Table 3.6: Initialization, solving, and total time for selected instances where greedy or path initialization performs well. The "Percent Init" column is the initialization time as a percentage of the total solving time.

In the first two examples, instances of mining data, greedy and path initialization take approximately twice as long as simple initialization, and they result in a reduced solving time and a reduction in the number of mergers. For Pit2, the additional initialization time is more than offset by the reduced solving time, and it results in a faster overall time. For mine1, greedy and path initialization reduce the solving time but not sufficiently to offset the added overhead of the more complex initialization.

Notice that for mining problem instances, greedy and path initialization are virtually identical in terms of initialization time and mergers during solving. This is expected because the arcs in the mining graphs that represent dependencies between the nodes have infinite capacity. Therefore, path initialization will not encounter a node without a

single arc with sufficient capacity to discharge the excess. Therefore, it will behave the same as greedy.

In the next instance, line-moderate, simple and path initialization take approximately the same length of time. This is because the capacity of the arcs into the source-adjacent nodes greatly exceeds the capacity of any arcs out of the nodes. Therefore, path initialization is stopped immediately and is effectively the same as simple initialization: both only saturate the source and sink arcs. This can be seen in the identical number of mergers. On the other hand, greedy initialization takes considerably longer than the other two. Although this results in a modest reduction in the number of mergers performed, the solving time is actually higher.

The final instance, genrmf-long, is consistent with our hypothesis; path and greedy initialization take slightly longer than simple initialization, but both spend about 10% less time in Phase I compared to simple initialization. The overall running time for path and greedy is less, and both perform fewer mergers as well.

3.6.2 Initial Labeling

As we see from the best-case heuristics in Table 3.5, the simple, constant labels are usually the best. The distance to deficit (or sink) is superior in longer graphs: genrmf-long, rlg-long, and line-moderate.

Distance to deficit or sink labeling is rather expensive compared to simple initialization. It requires a pass over all of the nodes to initialize their distances, $O(n)$, and the breadth-first search takes $O(m)$. We see in Table 3.6 above, that even simple initialization when combined with distance to deficit/sink labeling can take up to half of the total solving time.

3.6.3 Root Label and Normalization

We tested with three combinations of root label selection and normalization methods: highest-label (using immediate normalization), lowest-label with immediate normalization, and lowest-label with delayed normalization (see Table 2.1). As we can see in Table 3.5, the highest label pseudoflow algorithm is typically better than the lowest label algorithm

with immediate normalization, and delayed normalization is never the best choice.[10] However, in the case of the AK graph, the performance of lowest label and highest label is indistinguishable. Delayed normalization (for the lowest label algorithm) is never the best.

We believe that the superior performance of the highest label pseudoflow algorithm is similar to the superior performance of the highest label push-relabel algorithm. Goldberg and Cherkassky [GC97] suggested that "nodes on the source side of a cut are more likely to be processed than other nodes." The highest label algorithm tends to generate gaps which are recognized by the gap relabeling heuristic and causes the relevant nodes to be relabeled to n.

For the pseudoflow algorithm, the highest label algorithm tends to generate gaps that result in individual strong branches being pruned earlier, rather than waiting for a gap to exist between all strong nodes and the weak nodes, as happens with the early termination rule for the lowest label algorithm. Also, the highest label algorithm will tend to work with strong branches on the source side of the cut before processing lower-labeled branches closer to the sink (on the other side of the cut). On the other hand, the lowest label pseudoflow algorithm will tend to operate on branches on the sink side of the cut and to deliver excess to the sink-adjacent nodes as early as possible.

3.6.4 Branch Management

Table 3.5 shows that either the LIFO or wave branch management policy is the best-case heuristic in all of the modest-sized instances, with wave being superior in six out of ten cases. FIFO is never the best.

In Table 3.7 we look at the modest-sized instances using the same best-case heuristics from Table 3.5, and we include all three branch management heuristics. Each cell contains the execution time (in seconds) as well as the relative performance (best is 100).

We can see that wave is the best choice overall. In the worst case (Sched), wave is only 20% worse than the best heuristic (LIFO), and in all other cases, it is either the best or nearly indistinguishable from LIFO.

[10] The occurrences of "lowest" in Table 3.5 correspond to lowest label with immediate normalization—see Table 2.1.

Instance	LIFO	FIFO	Wave
AC	1.723	1.647	1.443
	119	114	100
genrmf-long	0.693	0.703	0.695
	100	101	100
genrmf-wide	3.812	3.577	3.487
	109	103	100
line-moderate	2.56	2.58	2.53
	101	102	100
rlg-long	0.608	0.606	0.594
	102	102	100
rlg-wide	8.01	1.558	1.496
	535	104	100
Pit2	1.56	2.08	1.58
	100	133	101
Sched	4.48	5.28	5.36
	100	118	120
Mine	7.07	6.72	6.47
	109	104	100

Table 3.7: Comparison of branch management heuristics on modest-sized instances.

3.6.5 Search Order

As we expected, pre-order searches are superior to post-order, which never appears in a best-case heuristic combination. A very dramatic example of this can be seen in the table below. It shows the simple-const-highest-lifo pseudoflow solver with pre- and post-order searches for a genrmf-long problem instance ($x = 16$, i.e. $n = 2^{16}$).

Search Order	Cut Time	Num Mergers	Num Pushes	Push/Merger
pre-order	4.00	318,019	2,673,206	8.41
post-order	73.51	734,009	73,057,320	99.53

The post-order search resulted in twice as many mergers, but the average length of the pushes during the mergers (the number of pushes per merger) was an order of magnitude larger than for the pre-order search.

3.6.6 Global Relabeling

Contrary to the experiences of most authors implementing push-relabel, global relabeling does not seem to be important for pseudoflow. Global relabeling was the best for rlg-long

and cheriyan. For rlg-long, the difference between global relabeling (with a frequency $4n$) and no global relabeling is insignificant—less than 1%. For cheriyan, the difference between global relabeling (with a frequency n) and no global relabeling is 19%, which is significant, but not the order of magnitude difference that previous authors found for the push-relabel algorithm.

That said, we did select a relabeling frequency of $4n$ for the pfs-C standard heuristic combination. It was superior to no relabeling in only half of the instances, but it was superior in a few crucial cases such as the line-moderate graph where no relabeling actually runs slower than the Dinic solver.

So, for some heuristic combinations (e.g., those used with rlg-long, cheriyan, and pfs-C), global relabeling can be a benefit. However, in the majority of cases, the best heuristic combination did not use global relabeling.

Initially, we assumed that performing global relabeling more frequently would always result in fewer operations being performed, but that the time spent performing global relabeling might outweigh the performance benefit. We did not directly collect data on the time spent performing global relabeling, but we can look at the number of operations performed by the algorithm and the total running time.

Table 3.8 contains data for two problem instances solved with all five relabeling frequencies we tested with. For each, we show the number of mergers, number of pushes, number of arcs scanned, the time to establish the cut, and the number of global relabeling operations that were performed.

For the first instance (genrmf-wide), we can see that performing global relabeling most frequently (every $n/2$ relabel operations) actually results in more mergers, pushes, and scans compared to only performing it every n relabel operations. Performing no global relabeling provides the fastest performance even though it performs more pushes and arc scans than the $2n$ frequency. It performs more work (pushes and scans) but runs faster, most probably due to the overhead of the global relabeling paid by the $2n$ relabel heuristic.

The second instance (genrmf-long) fits our initial assumptions better: more frequent global relabel operations result in fewer operations but a slower running time, most likely due to the overhead of global relabeling. Notice that no relabels are performed in the $4n$

		none	$n/2$	$1n$	$2n$	$4n$
genrmf-wide	mergers	149,681	250,248	186,849	164,691	168,981
	pushes	4,664,416	3,754,272	3,251,837	3,769,871	4,342,643
	scans	5,487,889	7,258,556	5,047,986	4,794,707	5,631,558
	cut time	4.84	11.68	6.36	5.47	5.41
	relabels	0	27	8	4	2
genrmf-long	mergers	137,904	136,915	137,059	137,153	137,904
	pushes	299,491	284,785	286,284	287,717	299,491
	scans	1,908,573	1,523,850	1,557,405	1,598,801	1,908,573
	cut time	0.92	1.55	1.21	1.07	0.93
	relabels	0	4	2	1	0

Table 3.8: Effects of global relabeling performed at various frequencies.

case; the solver completes before the first global relabeling operation.

3.6.7 Flow Recovery—Phase II

As mentioned in Section 2.2.3, flow recovery is expected to be a small percentage of the total solution time. This is generally true for both the pseudoflow algorithm and push-relabel, as shown in Table 3.9. The table shows the total time to compute the maximum flow, the time spent performing flow recovery, and the percentage of the total time that was spent in flow recovery.

As a percentage of the solution time, flow recovery for the pseudoflow solver is faster than push-relabel in all but two cases: cheriyan and rlg-long. In the case of rlg-long, the pseudoflow solver spends 40.57% of its time performing flow recovery compared to only 18.35% for push-relabel—a difference of 22%. However, in the worst case for the push-relabel algorithm performing flow recovery (line-moderate), push-relabel spends 56.58% of the time in flow recovery versus 10.89% for the pseudoflow solver–a difference of 45%. Therefore, both algorithms have instances where flow recovery takes 40% or more of the solving time, but the worst case for the pseudoflow solver is better than the worst case for the push-relabel solver.

3.6.8 Simplex

As expected, the simplex pseudoflow algorithm performs poorly. Table 3.10 contains some sample runs comparing the lowest label algorithm using the normal (pseudoflow) merger

Instance	Solver	Total Time	Flow Recovery Time	Percent of Total
AC	HL	3.89	0.337	8.67%
	pfs-EB	1.752	0.117	6.68%
AK	HL	10.414	0.014	0.13%
	pfs-EB	0.036	0.002	5.56%
cheriyan	HL	53.196	0.156	0.29%
	pfs-EB	2.554	0.054	2.11%
genrmf-long	HL	0.692	0.112	16.18%
	pfs-EB	0.788	0.073	9.26%
genrmf-wide	HL	6.831	1.798	26.32%
	pfs-EB	3.688	0.087	2.36%
line-moderate	HL	2.158	1.221	56.58%
	pfs-EB	2.829	0.308	10.89%
rlg-long	HL	0.436	0.080	18.35%
	pfs-EB	1.363	0.553	40.57%
rlg-wide	HL	1.375	0.091	6.62%
	pfs-EB	1.921	0.053	2.76%

Table 3.9: Flow recovery times for push-relabel (HL) and pseudoflow(pfs-EB) on modest-sized instances. (Mining and scheduling data are omitted because those applications are only concerned with minimum cuts.)

procedure and the simplex merger procedure. The poor performance of the simplex solver is due to the increased branch size, as indicated by the number of pushes per merger performed by the simplex solver. This is expected since the simplex solver only performs one split operation per merger, whereas the pseudoflow merger procedure can perform multiple splits, creating a number of smaller branches.

	Time		Pushes per Merger	
	pseudo	simplex	pseudo	simplex
genrmf-long	46.59	68.27	114.2	171.0
genrmf-wide	27.85	50.58	98.1	193.1
line-moderate	8.53	25.57	121.1	695.8
rlg-long	29.56	36.62	16.1	29.0
rlg-wide	28.81	48.66	39.5	65.1

Table 3.10: Sample solution times and number of pushes for simplex and lowest-label pseudoflow solvers on modest-sized instances.

3.7 Conclusion

Although the push-relabel algorithm has a slightly better theoretical complexity, we have shown that our implementation of the pseudoflow algorithm performs very well in practice. The performance of the pseudoflow solver is better in many cases than the more mature push-relabel implementation. This is significant for a first implementation of a new algorithm compared to a very mature implementation of a comparatively old and well-studied algorithm.

Many of the heuristics for the pseudoflow algorithm demonstrate significant improvements in the performance of the implementation even though they have no theoretical advantage. In particular, the highest label algorithm and the early termination rules (both new in this work) deliver superior performance for the pseudoflow family of algorithms. The heuristics allow the implementation to be easily tuned for a given application.

Chapter 4

Parametric Analysis

In this chapter, we present a new type of maximum flow/minimum cut problem where the capacities of the arcs are functions a single parameter. Gallo, Grigoiadis, and Tarjan [GGT89] presented a specific form of parametric maximum flow problem and showed how the push-relabel algorithm can be used as a subroutine to solve this parametric problem efficiently. Hochbaum [Hoc97] showed how the pseudoflow algorithm can also be used to solve these same types of problems efficiently.

Prior to this work, neither of these parametric algorithms had been implemented. The practical performance of the parametric algorithms had never been compared to simple, brute force application of the nonparametric maximum flow implementations.

In the previous chapter, we show that that our implementation of the pseudoflow algorithm is very competitive with the best implementation of the push-relabel algorithm for nonparametric problems. In this chapter, we present the experimental results of our implementations of the parametric push-relabel and pseudoflow algorithms compared to their nonparametric counterparts.

Our results show that both of the parametric algorithms are superior to their nonparametric counterparts. Both implementations execute considerably faster and require many fewer operations to solve parametric instances for a series of parameter values.

4.1 Introduction

Sensitivity analysis is well developed for linear programming and network flows (e.g., [BT97]). We can use sensitivity analysis to solve a number of similar problems that

differ only in the value of a single parameter without solving each individual problem from scratch—i.e., we can use information from one solution to aid in the solution of subsequent instances.

When solving linear programming problems, we can use sensitivity analysis to observe how the optimal solution is affected by changes to the value of a single parameter. For example, we might want to know how the value of the objective function changes as the value of the parameter changes, or we might want to know the range of parameter values for which the basis remains optimal.

There are applications of the maximum flow (or minimum cut) problem where the arc capacities are not fixed, but rather the capacities are functions of a single parameter. We might wish to know how the value of the maximum flow changes as the value of the parameter changes or how the minimum cut (or source set of the minimum cut) changes as the value of the parameter changes. Also, we might have a set of values for the parameter and wish to solve for the maximum flow (or minimum cut) for each of the parameter values.

Gallo, Grigoiadis, and Tarjan [GGT89] define a specific type of parametric network (see below) where the capacities of the source and sink arcs are functions of single parameter. Let $G_\lambda(V, A)$ be a parametric network where the capacities are functions of a real-valued parameter λ. Denote the capacity of an arc (i, j) as a function of the parameter value λ by $c_{ij}(\lambda)$ and assume the functions have the following properties:

- $c_{sv}(\lambda)$ is a monotone, non-decreasing function of λ for all $v \neq t$.
- $c_{vt}(\lambda)$ is a monotone, non-increasing function of λ for all $v \neq s$.
- $c_{uv}(\lambda)$ is constant for all $u \neq s$ and $v \neq t$.

As λ increases, the capacities of the arcs out of the source can only increase, and the capacities of the arcs into the sink can only decrease. The capacities of the other arcs remain constant.

At first glance, this structure of $G_\lambda(V, A)$ and the properties of $c_{uv}(\lambda)$ might appear to be artificial and restricted. In fact, there are a large number of applications that can be made to fit into this structure (see Section 4.2). Once formulated in this structure, these

problems can be solved efficiently for a sequence of values of λ by exploiting properties of the network structure and properties of certain algorithms to solve the maximum flow problem.

Let S_λ be the minimal source set[1] of a minimum s-t cut in G_λ. For a sequence of values of λ, $\lambda_1, \lambda_2, \ldots, \lambda_k$, we say that the minimal source sets of the minimum cuts are *nested* if $S_{\lambda_1} \subseteq S_{\lambda_2} \subseteq S_{\lambda_3} \ldots \subseteq S_{\lambda_k}$.

Lemma 4.1.1 *Given an ordered sequence of parameters, $\lambda_1 < \lambda_2 < \ldots < \lambda_k$, the minimal source sets for the s-t minimum cuts for these values are nested—i.e., $S_{\lambda_1} \subseteq S_{\lambda_2} \subseteq S_{\lambda_3} \ldots \subseteq S_{\lambda_k}$.*

This property was observed by a number of authors including Eisner and Severance [ES76] and Stone [Sto78], and it was proven in the context of the parametric push-relabel algorithm by Gallo, Grigoiadis, and Tarjan [GGT89].

Due to this property, the following is clear:

Lemma 4.1.2 *There are at most $n-1$ possible, distinct s-t minimum cuts in G_λ regardless of the number of values of λ.*

Based on the network defined by Gallo, Grigoiadis, and Tarjan, Hochbaum [Hoc97] defines the following two types of parametric analysis that can be performed:

Definition 4.1.3 Simple sensitivity analysis*: given an ordered series of values of λ, compute the maximum flow (or minimum cut) in G_λ for each value of λ.*

Definition 4.1.4 Complete parametric analysis*: compute the breakpoint values of λ where the minimal source set of the minimum cut in G_λ changes.*

It is well known that for nonparametric graphs, the minimum cut problem is related to the maximum flow problem (e.g., [AMO93]). Given a maximum flow (specified by the flow

[1]Note that Gallo, Grigoiadis, and Tarjan [GGT89] discuss this property and the lemma that follows in terms of "the minimum cuts produced by the algorithm" because the authors define a cut as the partition of nodes rather than the set of arcs between the partition of nodes, as we did in Chapter 1. However, since the minimum cut (set of arcs) for a given value of λ is not necessarily unique, the source set is not necessarily unique. Hochbaum [Hoc97] clarifies this by specifying "the minimal source set" of the minimum cut, and we adopt the same terminology.

values on every arc in the graph), it is easy determine the minimum cut. Therefore, the minimum cut problem is often solved using a maximum flow algorithm. However, given only a minimum cut (specified by the set of saturated arcs between the source set and sink set), the worst-case complexity of finding the corresponding maximum flow is comparable to solving the maximum flow problem without any prior knowledge of the minimum cut.

In fact, many of the parametric applications in the literature are parametric minimum cut problems (e.g., see Section 4.2). In these cases, parametric maximum flow algorithms are used to find a maximum flow from which a minimum cut or the source set of a minimum cut can easily be found. As we shall see, some parametric algorithms can identify the minimum cut for a given parameter value without identifying a feasible maximum flow.

Gallo, Grigoiadis, and Tarjan presented algorithms for solving these particular types of parametric maximum flow problems based on the push-relabel algorithm. Using the dynamic trees with the FIFO variant of push-relabel, the algorithm can perform simple sensitivity analysis on $O(n)$ parameter values with the same worst-case complexity as a single run of the push-relabel algorithm. Gallo, Grigoiadis, and Tarjan also showed that if the capacity functions are restricted to being linear, the parametric maximum flow algorithm can be used as a subroutine to perform complete parametric analysis with the same worst-case complexity as a single run of the push-relabel algorithm.

Hochbaum [Hoc97] showed how the pseudoflow algorithm can also be used to solve these problems efficiently. The parametric pseudoflow algorithm can perform simple sensitivity analysis on $O(m \log n)$ parameter values with complexity equal to that of a single run of the pseudoflow algorithm. The parametric pseudoflow algorithm can also be used to perform complete sensitivity analysis with the same complexity as a single run, assuming the capacity functions are linear.

In order to perform simple sensitivity analysis given a parametric network in the form of $G_\lambda(V, A)$ described above and an ordered series of values for λ, we have the choice of either creating separate nonparametric graphs for each value of λ and solving them with a nonparametric solver, or we could use a parametric algorithm in the hope that it would run faster. If we choose to use a parametric solver, there is a choice of possible parametric solvers. In this chapter, we consider two possible parametric algorithms: the push-relabel

based algorithm of Gallo, Grigoiadis, and Tarjan and the pseudoflow based algorithm of Hochbaum.

In the remainder of this chapter, we present some applications that can be formulated as this type of parametric network, and we investigate the performance of the alternatives facing us.

4.2 Applications

There are a number of applications for parametric maximum flow/minimum cut algorithms. In fact, most of the applications in the literature are parametric minimum cut problems. In these cases, the parametric maximum flow algorithms are used to find a maximum flow from which the minimum cut can easily be found. This means that for each parameter value, we produce additional information (e.g., flow values) beyond the minimum cut required by the application. When we discuss the particular algorithms in the following sections, we will see that there is a slight difference in their capabilities with regards to cuts versus flows.

In this section, we describe the open-pit mining and image segmentation applications, for which we present experimental results later in the chapter. We also briefly describe a number of additional applications from the literature.

- Open-pit mining: instances of the open-pit mining problem (e.g., see Johnson [Joh68]) are naturally suited for this type of parametric analysis. In the open-pit mining problem, a volume of earth is represented as a series of blocks. In order to remove a block (to gain its economic value), all the of the blocks in a cone above the block must also be removed. Each block of ore in the mine is represented as a node in the graph. There is an arc from a node to the nodes that represent blocks immediately above the block. These arcs have infinite capacity so that a minimum cut (of finite capacity) will not include any of these arcs and thus ensure that we remove all of the blocks above the given block. In other words, the set of nodes representing blocks that we extract defines a closure in the graph.

  The economic value of block and the cost to remove it are represented by a pair of arcs. Each node has a parametric arc to both the source and the sink. The

capacities of these ares are linear functions: $c_{sv}(\lambda) = \max(0, a + b\lambda)$ and $c_{vt}(\lambda) = -\min(0, a+b\lambda)$, where λ represents the value of a unit of the ore being mined, $b \geq 0$ represents the quantity of ore in a block, and $a \leq 0$ represents the cost to extract the ore from the block. Solving a series of minimum cut problems with a series of values of λ represents a simulation of the effects of a series of values for the price of the ore.

- Image segmentation: Hochbaum [Hoc01] presented an application of complete parametric analysis for image segmentation, which can also be solved using simple parametric analysis. An image is represented by a grid of pixels. Each pixel takes on a single integer value from zero to k that represents the color or grayscale value of the pixel. The image is transmitted and degraded by noise. The goal is to modify the pixel values to conform with prior information about the image by minimizing a penalty function composed of two terms: the first term penalizes the deviation from the initial pixel value as a convex function, and the second term penalizes the difference in colors between neighboring pixels. Hochbaum showed how this problem can be solved by parametric minimum cut to identify new values for the pixels that minimize the total penalty.

  Hochbaum constructs a parametric graph $G_\lambda = (V, A)$ where each pixel is represented as a node $j \in V$. Let $p_j(\lambda)$ represent the penalty function for deviation of pixel j from its original value. This is represented in the graph as two arcs: one from s to the node with $c_{sj}(\lambda) = -\min\{0, p'_j(\lambda)\}$, and one from the node to the sink with $c_{jt}(\lambda) = \max\{0, p'_j(\lambda)\}$, where $p'_j(\lambda)$ is the derivative or subgradient of p_j at λ. Each node has arcs to each of its two, three, or four neighbors with a constant capacity representing a penalty for discontinuities in terms of separation of color between adjacent pixels. The parameter values take the same values as the pixels: integers from 0 to k representing the spectrum of colors. For each value of λ, the source set of the minimum cut in G_λ identifies the set of pixels that should have a value less than or equal to λ, and the sink set represents those pixels that should have a value greater than λ.

Gallo, Grigoiadis, and Tarjan [GGT89] presented a large number of applications for parametric maximum flow/minimum cut including the following:

- Flow sharing: consider a network with k source nodes and a single sink. Define the utilization u_i of a source s_i as the flow out of source s_i, and suppose each source has a positive weight w_i associated with it. There are a number of possible optimization problems called "flow sharing" based on the ratio u_i/w_i:
  Perfect sharing: Find a maximum flow such that u_i/w_i is equal for all i.
  Maximin sharing: Find a maximum flow that maximizes the smallest value of u_i/w_i.
  Minimax sharing: Find a maximum flow that minimizes the largest value of u_i/w_i.
  Optimal sharing: Find a maximum flow that simultaneously maximizes the smallest value and minimizes the largest value of u_i/w_i.

- Maximum density subgraph: Goldberg [Gol84] described the problem of finding the maximum density subgraph in an undirected graph. The density of a subgraph is the ratio of the number of edges to number of nodes in the subgraph. The objective is to identify a non-empty subgraph with maximum density.

- Strength of a network: Cunningham [Cun85] described the problem of computing the strength of a network. The goal is to disconnect a set of nodes from the source node by removing a subset of edges in the graph. Associated with each node is a value attributed to having the node reachable from the source, and associated with each arc is a cost. The ratio of the total cost to the total value is the cost per unit reduction, and the objective is to minimize this ratio over all subsets of arcs.

Other more recent applications include:

- Baseball elimination: there are a number of questions related to a baseball team's standing at the end of a season and whether the team has been eliminated from the play-offs. Schwartz [Sch66] showed that a maximum flow calculation can determine when a team has been eliminated from first place. Gusfield and Martel [GM92] showed how parametric maximum flow can be used to compute the minimum number of games a team must win to avoid elimination, and McCormick [McC99] improved the complexity result.

- Bar-Noy and Kortsarz [BNK98] used parametric maximum flow to solve a variant of the selection problem called the 2-neighbor program. This, in turn, is used to approximate the minimum color sum of a bipartite graph in which the colors of the vertices of a graph are such that the sum (or average) of all assigned colors is minimum.

- Margot, Queyranne, and Wang [MQW00] discussed a scheduling algorithm for N jobs on a single machine with a partial ordering. The authors used parametric maximum flow to identify a decomposition of the jobs such that an optimal schedule is obtained from the schedules of the job subsets in the decomposition.

- Frederickson and Solis-Oba [FSO97] presented algorithms for computing the robustness function of a matroid. For traversal matroids, they presented an algorithm based on parametric maximum flow that improves on the time complexity of their algorithm for arbitrary matroids.

- Fleischer [Fle01] presented algorithms for dealing with flows over time, which are also referred to as dynamic network flows in some papers. The author used parametric maximum flow to show that the quickest transshipment problem with a single sink (also known as the evacuation problem) can be solved with k maximum flow computations, where k is the number of source nodes in the network. The author also used parametric maximum flow to solve the single sink, universally quickest transshipment problem in the same asymptotic time as one maximum flow problem using the parametric push-relabel algorithm.

Note that most of the applications listed above (and all of the ones we tested with) are interested in identifying the minimum cut rather than the maximum flow; parametric maximum flow is used to find the minimum cut.

4.3 Simple Parametric Push-Relabel Algorithm

Gallo, Grigoiadis, and Tarjan [GGT89] presented an algorithm called the parametric push-relabel algorithm for solving the simple sensitivity analysis problem.[2] Given a parametric

[2]Gallo, Grigoiadis, and Tarjan also presented an algorithm for complete parametric analysis.

network $G_\lambda = (V, A)$ with source s, sink t, capacity functions $c_{uv}(\lambda)$, and an ordered sequence of parameters, $\lambda_1 < \lambda_2 < \ldots < \lambda_k$, the parametric push-relabel algorithm operates by applying the push-relabel algorithm to the parametric network G_λ with successive values of λ_i. Once the push-relabel algorithm has computed the maximum flow and the source set of the minimum cut S_{λ_i}, we can apply the next parameter value λ_{i+1} to the capacities, adjust the flows with respect to the new capacities, and continue solving (using the push-relabel algorithm) with the existing values for the distance labels and the adjusted flow values.

The algorithm is initialized for the first value of λ like the nonparametric push-relabel algorithm by setting the distance labels $d(v)$ and the initial preflow f as follows: $d(s) = n$, $d(v) = 0\ \forall v \neq s$, and the arcs out of the source are saturated, $f(s, v) = c_{sv}(\lambda_1)$, $f(u, v) = 0$ $\forall u \neq s$.

We solve for the maximum flow in G_{λ_1} using the push-relabel algorithm. We solve subsequent instances of G_{λ_i} using the following steps:

After solving for the maximum flow for the previous parameter value λ_{i-1}, we adjust the capacities and flows on the source and sink arcs. For arcs out of the source, we set $f(s, v) = c_{sv}(\lambda_i)$.[3] However, if $d(v) \geq n$, we know that v is part of the source set of the minimum cut, and by the nesting property, we know it will be in the source set for all successive values of λ. This implies that there is no output for any additional flow from s into v. Therefore, we only adjust $f(s, v)$ if $d(v) < n$.

For arcs into the sink, the new capacity may be less than the existing flow. Therefore, we set $f(v, t) = \min\{f(v, t), c_{vt}(\lambda_i)\}$.

We solve for a new maximum flow with the existing label values $d(v)$ and new values for $f(u, v)$ and $c_{uv}(\lambda_i)$. After solving for a maximum flow in G_{λ_i}, the minimal source set S_{λ_i} of the minimum cut is the set of all nodes with $d(v) \geq n$.

procedure parametricPushRelabel($V, A, s, t, c_{uv}(\lambda), \{\lambda_1 \ldots \lambda_k\}$)**:**
  $d(s) = n$, $d(v) = 0\ \forall v \neq s$, $f = 0$
  $\langle\langle$*first iteration*$\rangle\rangle$
  foreach neighbor v of s**:**
    $f(s, v) \leftarrow c_{sv}(\lambda_1)$

[3] In Gallo, Grigoiadis, and Tarjan, this is specified as $\max\{f(s, v), c_{sv}(\lambda_i)\}$, but it is easy to see that this is always $c_{sv}(\lambda_i)$.

$\langle\langle$*solve maximum flow problem with push-relabel algorithm*$\rangle\rangle$
call push-relabel$(V, A, s, t, c_{uv}(\lambda_1), f, d)$
$S_{\lambda_1} \leftarrow \{v : d(v) \geq n\}$

for $i \leftarrow 2$ **to** k**:**
  foreach neighbor v of s, $v \notin S_{\lambda_{i-1}}$**:**
    $f(s, v) \leftarrow c_{sv}(\lambda_i)$
  foreach neighbor v of t**:**
    $f(v, t) \leftarrow \min(f(v, t), c_{vt}(\lambda_i)$
  $\langle\langle$*solve maximum flow problem with push-relabel algorithm*$\rangle\rangle$
  call push-relabel$(V, A, s, t, c_{uv}(\lambda_i), f, d)$
  $S_{\lambda_i} \leftarrow S_{\lambda_{i-1}} \cup \{v : d(v) \geq n\}$

Gallo, Grigoiadis, and Tarjan [GGT89] proved the correctness of the algorithm, which can be summarized in the following theorem.

Theorem 4.3.1 *During the execution of the parametric push-relabel algorithm, after updating the capacities and flows on the arcs for a new parameter value, the modified f is a preflow, and the distance labels d are valid for the modified f—i.e., $d(v) \leq d(w) + 1$ for every residual arc (v, w).*

After solving the first instance, the additional time required to solve the subsequent instances is composed of the time to adjust the flows and capacities and the time to invoke the push-relabel solver. For each iteration, the time to adjust the capacities and flows of the source and sink arcs is $O(n)$. The total time to output the source sets (after calling push-relabel on each instance) is $O(n)$. Therefore, for k values of the parameter, the time required for these steps is $O(kn)$.

Gallo, Grigoiadis, and Tarjan [GGT89] analyzed the additional time spent solving the $k - 1$ instances of G_λ after solving the first instance. They analyzed three variants of the parametric push-relabel algorithm (generic, FIFO, and FIFO with dynamic trees) using methods analogous to those used by Goldberg and Tarjan [GT86] for the nonparametric push-relabel algorithm. The analysis focuses on the number of non-saturating pushes using the same potential function Φ (the sum of the labels of the active nodes) used by Goldberg and Tarjan. The generic and FIFO algorithms perform $O(n^2)$ additional non-saturating pushes per parameter value, and the FIFO algorithm with dynamic trees performs $O(m \log(n^2/m))$ non-saturating pushes per parameter value. In all cases, the

total time spent solving k instances when $k = O(n)$ is the same as a single run of the corresponding push-relabel algorithm

Gusfield and Tardos [GT94] analyzed the parametric push-relabel algorithm using the highest label variant of push-relabel with and without dynamic trees by combining the analysis of Cheriyan and Maheshwari [CM89] for nonparametric highest-label with the parametric techniques of Gallo, Grigoiadis, and Tarjan. For both forms of the highest label algorithm, the total time spent solving $O(n)$ instances is the same as a single run of the corresponding variant of the highest label algorithm.

In Table 4.1 below, we show the complexity for a single run, k runs, and $O(n)$ runs for each variant of the parametric push-relabel algorithm. For all variants, the complexity for $O(n)$ runs is the same as a single run. This can be summarized in the following theorem from Gallo, Grigoiadis, and Tarjan [GGT89].

Theorem 4.3.2 *Using dynamic trees for the push-relabel algorithm, the overall worst-case complexity for the parametric push-relabel algorithm for k parameter values is $O(m(n + k)\log(n^2/m))$.*

Variant	Single Run	k runs	$O(n)$ runs
Generic [GGT89]	n^2m	$n^2m + n^2k + kn$	n^2m
FIFO [GGT89]	n^3	$n^3 + n^2k + kn$	n^3
Highest Label [GT94]	$n^2\sqrt{m}$	$n^2\sqrt{m} + (n^{1.5}\sqrt{mk} + n^{1.5}k) + kn$	$n^2\sqrt{m}$
Dynamic Tree FIFO [GGT89]	$nm\log(n^2/m)$	$nm\log(n^2/m) + km\log(n^2/m) + kn$	$nm\log(n^2/m)$
Dynamic Tree Highest Label [GT94]	$nm\log(n^2/m)$	$nm\log(n^2/m) + kn\log(n^2/m) + kn$	$nm\log(n^2/m)$

Table 4.1: Complexities for parametric push-relabel algorithms. The complexity expressions for k runs are composed of three terms: the time spent solving the first instance with the indicated push-relabel algorithm, the additional time spent solving the subsequent instances, and the time to update the capacities and flows for each value of λ.

4.3.1 Solving for Maximum Flow versus Minimum Cut

Many applications require finding a minimum cut rather than a maximum flow. Since there is no known algorithm to find the minimum cut that does not involve at least as much work as finding the maximum flow, maximum flow algorithms are typically used

to solve for minimum cuts. Given a maximum flow produced by such an algorithm, a minimum cut can be easily be identified.

The majority of applications of simple parametric analysis are actually only interested in identifying a minimum cut, the source set of a minimum cut, or the value of the minimum cut for each value of λ. Gallo, Grigoiadis, and Tarjan [GGT89] presented a variant of simple parametric analysis using the push-relabel algorithm that they called the *min-cut parametric algorithm.*

During each iteration of the parametric algorithm when the push-relabel algorithm is called to solve for a new maximum flow, we instead invoke the first phase of a two-phase push-relabel algorithm [GT86] which only regards nodes to be active if $e_f(v) > 0$ and $d(v) < n$. At the end of the first phase of the two-phase push-relabel algorithm, the result is a *maximum preflow* where the the nodes with excess have labels greater than or equal to n, which indicates that they have no path to t in G_f [GGT89].

After substituting the first phase of the two-phase algorithm for the complete single-phase algorithm, the rest of the parametric push-relabel algorithm is unchanged. The result of each iteration of this new parametric push-relabel algorithm is a (maximum) preflow, rather than a feasible flow. The minimal source set of the minimum cut S_{λ_i} is the set of nodes with $d(v) \geq n$. From this preflow, the minimum cut can be identified in linear time.

The following theorem follows from Gallo, Grigoiadis, and Tarjan [GGT89], although the authors do not explicitly state or prove such a theorem.

Theorem 4.3.3 *The min-cut parametric algorithm using the first phase of a two-phase push-relabel algorithm is correct, and the complexity is the same as using a single-phase push-relabel algorithm.*

Proof: We prove this by induction on the iteration number. After the first value of λ, f is a valid preflow and the labels $d(v)$ are valid because that is the result of the first phase of the two-phase push-relabel algorithm. By the inductive hypothesis, assume that the first $k-1$ iterations maintain a valid preflow f and valid labels $d(v)$.

After adjusting the capacities and flows at the beginning of iteration k, f is a still a valid preflow because the adjustment procedure only increases the excess of nodes (other

than s and t), and the flows on each arc are adjusted such that the capacity constraints are maintained. The labels $d(v)$ are valid for the same reason they were when using the single-phase algorithm (see Theorem 4.3.1): the only new residual arcs created by adjusting the capacities and f are of the form (s, v) for nodes with $d(v) \geq n$ and (v, s) for nodes with $d(v) < n$. After calling the first phase push-relabel, f is a valid preflow, and the labels are still valid. Therefore, the algorithm is correct throughout its execution.

The complexity of the first phase of a two-phase algorithm is identical to that of a single-phase push-relabel algorithm [GT86], and the rest of the parametric algorithm is unchanged. The labels do not decrease during the execution of the algorithm, and their maximum value is still bounded by $2n-1$. Therefore, the complexity of min-cut parametric algorithm is unchanged. ■

4.4 Simple Parametric Pseudoflow Algorithm

Hochbaum [Hoc97] showed how the pseudoflow algorithm can be used to solve simple and complete parametric analysis problems. The parametric extension of the pseudoflow algorithm achieves a similar, but slightly better, complexity result than that of the parametric extension to the push-relabel algorithm; the parametric pseudoflow algorithm can solve a simple parametric problem for k parameter values with the same complexity as a single run plus $O(kn)$. The parametric pseudoflow algorithm can solve $O(m \log n)$ instances with the same complexity as a single run.

Given a parametric network $G_\lambda = (V, A)$ with source s, sink t, capacity functions $c_{uv}(\lambda)$, and an ordered sequence of parameters, $\lambda_1 < \lambda_2 < \ldots < \lambda_k$, the parametric pseudoflow algorithm begins by constructing an initial pseudoflow using any of the initialization procedures in Section 2.6.1. The parametric pseudoflow algorithm proceeds by repeatedly invoking Phase I, once for each value of λ. Once Phase I is complete for a parameter value, the flows and capacities are adjusted for the next parameter value by increasing the flows on the arcs out of the source to match their new capacities and reducing the flows on the arcs into the sink to match their new capacities.

Hochbaum showed that nodes with labels greater than or equal to n are in the source set S_{λ_i} and will continue to be in the source set for all subsequent parameter values.

Therefore, we only need to update the capacities and flows of arcs from the source to those nodes that are not already in the source set—i.e., those nodes such that $d(v) < n$.

After adjusting the flows and capacities, the resulting pseudoflow may violate the normalized forest properties because positive excess may be created at source-adjacent nodes that are not the roots of branches. Therefore, we renormalize the entire forest by pushing the excess towards the root, splitting the branches as needed. This can be done by traversing each branch via a post-order traversal to renormalize from the leaves up to the root. As shown by Hochbaum, this procedure can be performed in $O(n)$ time.

After establishing a new normalized forest for the capacities $c_{uv}(\lambda_i)$, the pseudoflow solver is called again. The entire algorithm is shown below. Since the parametric algorithm can be used with the generic, lowest label, and highest label variants of the pseudoflow algorithm, we use the procedure *pseudoflowPhaseI(*V, A, s, t, c*)* which can be substituted for Phase I of any particular variant of the pseudoflow algorithm. Let *renormalize(*V, A, f, c, par*)* denote the procedure to renormalize the tree structure after adjusting the flows for a new parameter value.

procedure parametricPseudoflow($V, A, s, t, c_{uv}(\lambda), \{\lambda_1 \dots \lambda_k\}$)**:**
  $\langle\langle$*first iteration*$\rangle\rangle$
  call genericInit($V, A, s, t, c_{uv}(\lambda_1)$)
  call pseudoflowPhaseI($V, A, s, t, c_{uv}(\lambda_1)$)
  $S_{\lambda_1} \leftarrow \{v : v \text{ is strong}\}$

  for $i \leftarrow 2$ **to** k**:**
    foreach neighbor v of s, $v \notin S_{\lambda_{i-1}}$**:**
      $f(s, v) \leftarrow c_{sv}(\lambda_i)$
    foreach neighbor v of t, $v \notin S_{\lambda_{i-1}}$**:**
      $f(v, t) \leftarrow c_{vt}(\lambda_i)$
    call renormalize($V, A, f, c_{uv}(\lambda_i), par$)
    $\langle\langle$*execute Phase I of the pseudoflow algorithm*$\rangle\rangle$
    call pseudoflowPhaseI($V, A, s, t, c_{uv}(\lambda_i)$)
    $S_{\lambda_i} \leftarrow S_{\lambda_{i-1}} \cup \{v : v \text{ is strong}\}$

Hochbaum [Hoc97] proved the following theorems related to the correctness and complexity of the parametric pseudoflow algorithm.

Theorem 4.4.1 *During the execution of the parametric pseudoflow algorithm, after adjusting the capacities and flows for a new parameter value and renormalizing the branches, the properties of the normalized forest and node labels still hold.*

Theorem 4.4.2 *Using the highest and lowest label pseudoflow algorithms with dynamic trees, the overall worst-case complexity for the parametric pseudoflow algorithm for k parameter values is $O(mn \log n + kn)$.*

Corollary 4.4.3 *For $O(m \log n)$ parameter values, the overall worst-case complexity for the lowest and highest label algorithms using dynamic trees is $O(mn \log n)$—i.e., the same complexity as a single run.*

Note that this is better than the push-relabel algorithm which has the same complexity as a single run only for $k = O(n)$.

The improved complexity of the parametric pseudoflow algorithm compared to the parametric push-relabel algorithm is a result of branch structure maintained by the pseudoflow algorithm. In both the parametric push-relabel and parametric pseudoflow algorithms, adjusting the flows and capacities costs $O(n)$ time and creates $O(n)$ nodes with increased excess. With the push-relabel algorithm, the excess at the nodes results in additional non-saturating pushes or dynamic tree operations beyond those represented in the complexity of a single run. With the pseudoflow algorithm, this excess is moved to the roots of the branches in the renormalization procedure, which costs the same as the flow and capacity adjustment procedure, $O(n)$. After renormalization, the total number of merger operations does not increase—i.e., it is still bounded by $O(mn)$ regardless of the number of parameter values.

4.4.1 Solving for Maximum Flow versus Minimum Cut

In contrast to the parametric push-relabel algorithm that can provide the maximum feasible flow during each iteration (by using the single phase push-relabel algorithm), the parametric pseudoflow algorithm only computes a maximum pseudoflow at each iteration. From this pseudoflow, a minimum cut or the minimal source set of a minimum cut can be easily computed. However, in order to compute a maximum flow for each parameter value, the flow recovery procedure (Phase II) would have to be performed after computing the minimum cut, and this flow recovery procedure would need to be applied to a copy of the problem instance in order to leave the original instance intact to solve for the next parameter value. Recovering the flow would require $O(m \log n)$ time per parameter value.

For $O(n)$ parameter values, the overall complexity would be unchanged, however the practical performance would certainly be slower compared to not recovering the feasible flow for each parameter value.

4.5 Implementations

For our experiments, we look only at solving the simple parametric problem where we are given a parametric graph G_λ and a sequence of parameter values $\lambda_1, \ldots, \lambda_k$ and wish to find a minimum cut[4] for each value of the parameter. This section describes the software we developed for the experiments.

4.5.1 Solvers

Because there are no known implementations of the parametric push-relabel algorithm [Gol00], we could not test with an existing implementation and were required to write one. We developed a solver to perform simple parametric sensitivity analysis using the highest label implementation of Goldberg and Cherkassky [GC97] as the underlying push-relabel solver subroutine. The parametric pseudoflow solver was similarly implemented with the option of using either the highest label or lowest label pseudoflow solver as a subroutine.

In order to measure the benefits of the parametric algorithms performing simple parameteric analysis, we need to compare the performance of these algorithms to the performance of solving the problems without the benefit of the parametric algorithms. To do this, we take a parametric graph G_λ and convert it to a nonparametric graph by substituting one value of the parameter λ into the capacity functions $c_{uv}(\lambda)$ to compute an exact capacity value for each arc (u, v). This nonparametric graph can then be solved using either the push-relabel or pseudoflow solvers. This process is repeated for each value of λ.

To automate this process of converting graphs and solving them, we developed what we call an *iterative solver* that takes a parametric maximum flow instance G_λ and a series of parameter values $\lambda_1, \ldots, \lambda_k$ and solves for a minimum cut in G_{λ_i} repeatedly for each parameter value, using either the push-relabel or pseudoflow solver as a subroutine. Each

[4]We could solve for the maximum flow, but our applications are only concerned with minimum cut.

iteration initializes the solver from scratch, without any information about the flows or labels from the previous solution.

So, for each parametric graph and sequence of parameter values, we can test six possibilities: we can apply the parametric or iterative techniques to the push-relabel algorithm, the highest label pseudoflow algorithm, or the lowest label pseudoflow algorithm.

4.5.2 Workbench

We adapted the workbench software described in Section 3.2.3 to invoke the new parametric solvers and the iterative solver.

4.6 Testing Methodology

4.6.1 Testing Procedure

For each graph instance, we compared the performance of the parametric solvers to the iterative solvers. For both the parametric and iterative solvers, we report the sum of the times to establish the k minimum cuts for k parameter values. As before, we exclude the time to read the instances, and we report user CPU time as reported by the POSIX `times` system call [Pos88].

All testing was performed on the same Sun Microsystems computer described in Section 3.3.

4.6.2 Data Files

For testing we selected two application-based instance classes and one modified synthetic problem class. The applications we selected are discussed in general in Section 4.2. In this section we discuss the specific parameter values we tested with.

4.6.2.1 Mining

The data files we used were the same synthetic mining data we used for the nonparametric testing (see Section 3.4.2.5). These were converted to the parametric format such that $\lambda = 100$ coincides with the original data. We tested with nine values of λ: 50, 75, 90, 100, 120, 140, 160, 180, and 200. For a mining operation, this would represent scenarios where

the price of ore ranged from one half the current value ($\lambda = 50$) to twice the current value ($\lambda = 200$).[5]

4.6.2.2 Image Segmentation

We used two data files based on two digital images. Both images were reduced to 16 shades of grayscale before being converted into parametric graph instances. The first instance was derived from an image of 128 by 128 pixels. The resulting file had 16,384 nodes and 81,409 arcs. The second instance was based on an image of 229 by 210 pixels. The resulting file had 48,092 nodes and 239,573 arcs.

For both instances, we chose the quadratic function $p_j(\lambda) = (g_j - \lambda)^2$ for the penalty function related to the deviation of a pixel j from its original value, g_j. So for each pixel node j, $c_{sj}(\lambda) = 2\max(\lambda - g_j, 0)$ and $c_{jt}(\lambda) = 2\max(g_j - \lambda, 0)$. For the arcs between nodes that represent a penalty for discontinuities in terms of separation of color between adjacent pixels, we used the constant capacity $c = 4$.

Although image segmentation is an application of complete parametric analysis, we solved the problems with our simple sensitivity parametric solvers by iterating over the range of possible pixel values (excluding the minimum and maximum pixel values). Specifically, the parameter λ took the integer values from 1 to 14.

4.6.2.3 RLG Extra Wide

In many synthetically generated data files the out-degree of the source and the in-degree of the sink are limited. So, the number of parametric arcs would be limited if the network were converted to a parametric network. In order to create an interesting problem class, we used an idea from Badics and Boros [BB93] to create an "extra wide" version of the random level graph (RLG)—see Section 3.4.2.2. These are grid graphs where the nodes are arranged in rows and columns. The source is connected to every node in the first column, and the sink is connected to every node in the last column. We fixed the number of columns at four, and for a graph with 2^x nodes, there are $2^{(x-2)}$ rows.

We converted these graphs to parametric graphs by changing the original capacity

[5]The cost to extract the blocks and the amount of ore in each block are held constant—i.e., they are not parameterized.

values $c(u, v)$ of the source and sink arcs to linear parametric functions, $c_{uv}(\lambda) = \max(a + b\lambda, 0)$, where a and b are specific to each parametric arc. For each source or sink arc (u, v), we choose a random number r uniformly from $[0.2, 0.8]$. For source arcs (s, v), we set $a = r \cdot c(s, v)$ and $b = (c(s, v) - a)/100$. For sink arcs (v, t), $a = -r \cdot c(v, t)$, and b is the same as a source arc, $b = (c(v, t) - a)/100$. By construction, for $\lambda = 100$, the graph has the same capacities as the original, nonparametric graph. We tested with seven values of λ: 50, 75, 100, 125, 150, 175, and 200.

4.7 Results

For each problem type, we compare the performance of the following solvers.

iter-pr-hl: the iterative, highest label push-relabel solver.

param-pr-hl: the parametric, highest label push-relabel solver.

iter-pfs-lo: the iterative, lowest label pseudoflow solver.

param-pfs-lo: the parametric, lowest label pseudoflow solver.

iter-pfs-hi: the iterative, highest label pseudoflow solver.

param-pfs-lo: the parametric, highest label pseudoflow solver.

For the pseudoflow algorithm, we selected the best heuristic combination for the non-parametric solver identified in Chapter 3 and used that combination with the parametric pseudoflow solver. We tested both the highest and lowest label algorithms with this heuristic combination.

4.7.1 Mining

We tested with three instances of the mining data, *mine-54k*, *mine-108k*, and *mine-216k*, with 54,000, 108,000, and 216,000 nodes, and 410,000, 826,000, and 1,662,000 arcs respectively. The heuristic combination for the pseudoflow solvers used simple initialization, constant initial node labels, pre-order searches, wave branch management, and no global relabeling (simple-const-highest/lowest-pre-wave-0).

Table 4.2 shows the execution times (in seconds) to establish a minimum cut for the solvers running with mining data. For each iterative/parametric pair, the best time is shown in bold font followed by the ratio of the iterative time to the parametric time (i.e., a higher ratio indicates better performance for the parametric solver).

	mine-54k	mine-108k	mine-216k
iter-hl	12.73	30.50	77.70
param-hl	**3.97**	**9.44**	**22.97**
ratio	3.21	3.23	3.38
iter-pfs-lo	10.70	25.53	56.62
param-pfs-lo	**2.97**	**7.19**	**16.65**
ratio	3.60	3.55	3.40
iter-pfs-hi	8.30	18.98	41.14
param-pfs-hi	**2.17**	**4.98**	**10.98**
ratio	3.82	3.81	3.75

Table 4.2: Running times for parametric and iterative solvers for mining data with $\lambda \in \{50, 70, 90, 100, 120, 140, 160, 180, 200\}$.

From this we can see that the parametric solvers all run three to four times faster than the iterative solvers for all data instances. The parametric highest label pseudoflow solver is the fastest overall, and the lowest label pseudoflow solver is faster than the parametric push-relabel solver.

Table 4.3 shows the details of solving times, number of pushes, and number of mergers (for the pseudoflow algorithm) for each value of λ for the largest file, *mine-216k*. For the parametric solvers, these values represent the additional number of operations that were performed for the value of λ. For the iterative solvers, these values represent the number of operations to solve the instance from scratch. The last column contains the total count of all operations across the values of λ.

Solver	λ Value 50	70	90	100	120	140	160	180	200	Total
	Cut Time									
iter-hl	1.05	1.48	6.60	16.20	15.24	12.60	10.53	7.82	6.18	77.70
param-hl	1.62	1.82	5.57	7.47	2.27	1.15	1.05	1.00	1.02	22.97
ratio	0.65	0.81	1.18	2.17	6.71	10.96	10.03	7.82	6.06	3.38
iter-pfs-lo	0.60	1.14	5.99	10.92	7.40	7.44	7.73	8.09	7.31	56.62
param-pfs-lo	0.60	0.97	5.06	6.19	1.57	0.67	0.55	0.46	0.52	16.59
ratio	1.00	1.18	1.18	1.76	4.71	11.10	14.05	17.59	14.06	3.41
iter-pfs-hi	0.64	1.25	3.84	6.49	6.22	6.02	6.04	5.18	5.46	41.14
param-pfs-hi	0.62	0.84	2.92	3.44	1.30	0.54	0.41	0.38	0.46	10.91
ratio	1.03	1.49	1.32	1.89	4.78	11.15	14.73	13.63	11.87	3.77
	Number of Pushes									
iter-hl	103,910	432,378	2,231,036	5,726,267	5,891,398	5,007,518	4,416,187	3,403,341	2,703,797	29,915,832
param-hl	104,331	360,232	1,519,147	2,532,668	726,090	229,631	154,854	130,215	178,835	5,936,003
ratio	1.00	1.20	1.47	2.26	8.11	21.81	28.52	26.14	15.12	5.04
iter-pfs-lo	62,388	269,512	2,362,383	3,267,192	879,779	576,798	451,447	352,384	480,847	8,702,730
param-pfs-lo	62,388	411,316	2,259,816	1,667,805	277,695	145,243	119,573	106,541	93,449	5,143,826
ratio	1.00	0.66	1.05	1.96	3.17	3.97	3.78	3.31	5.15	1.69
iter-pfs-hi	55,268	224,220	1,047,909	1,747,241	1,492,912	1,525,106	1,443,368	1,150,545	1,225,700	9,912,269
param-pfs-hi	55,268	326,574	1,137,366	1,218,478	386,177	141,139	111,140	98,495	112,170	3,586,807
ratio	1.00	0.69	0.92	1.43	3.87	10.81	12.99	11.68	10.93	2.76
	Number of Mergers									
iter-pfs-lo	42,512	89,946	182,602	207,605	158,138	141,890	133,160	107,047	141,722	1,204,622
param-pfs-lo	42,512	49,818	97,292	40,642	10,684	8,592	6,623	5,377	4,382	265,922
ratio	1.00	1.81	1.88	5.11	14.80	16.51	20.11	19.91	32.34	4.53
iter-pfs-hi	44,330	101,900	230,107	297,976	315,845	327,575	339,364	286,511	308,365	2,251,973
param-pfs-hi	44,330	57,722	127,594	79,915	30,314	10,686	8,480	6,846	6,719	372,606
ratio	1.00	1.77	1.80	3.73	10.42	30.65	40.02	41.85	45.89	6.04

Table 4.3: Cut time, number of push operations and merger operations for parametric and iterative, push-relabel and pseudoflow solvers for mining data file *mine-216k*.

As expected, the operation counts for the subsequent values of λ when using any of the parametric solvers are generally substantially less than solving the problem from scratch. The parametric push-relabel algorithm performs only 1% more push operations (5.94 million versus 5.89 million) and takes only 42% longer (22.97 sec versus 16.20 sec) to solve the entire sequence compared to the most difficult single instances ($\lambda = 100, 120$). Likewise, the lowest label parametric pseudoflow solver performs only 57% more pushes (5.14 million versus 3.28 million) and takes only 52% longer (16.59 sec versus 10.92) than solving the most difficult single instance ($\lambda = 100$), and the highest label parametric pseudoflow solver takes 105% more pushes (3.57 million versus 1.75 million) and only 68% more time (10.91 sec versus 6.49) to solve the entire sequence.

So, all of the solvers can solve the entire sequence of nine parameter values with less than twice the effort required to solve the single worst instance in the sequence.

One anomaly in the results is that the parametric push-relabel solver is considerably slower than the iterative push-relabel solver for the first parameter value ($\lambda = 50$), even though the work (solving one instance) should be identical. This is a result of an implementation decision (described below) to alter the way in which the mining problem is modeled for the parametric push-relabel solver. This change sacrifices performance of the parametric push-relabel solver when operating with a single parameter value to achieve greater performance when processing subsequent parameter values.

For the parametric minimum cut problem with mining data, the parameter λ represents the value of one unit ore. Blocks have either positive or negative net economic value depending on the amount of ore in the block and the unit price of the ore. Positive blocks are modeled with a parametric arc (s, v), and negative blocks are modeled as a parametric arc (v, t). As λ increases, the values of the blocks increase, and negative blocks may become positive. In this event, the arc (v, t) must be replaced with (s, v).

Our parametric push-relabel solver is based on the code of Goldberg and Cherkassky [GC97]. However, the data structures used by that code did not lend themselves to efficiently converting an arc from a sink arc (v, t) into a source arc (s, v). In fact, early profiling revealed that this was causing a significant performance problem.[6]

[6]Since the parametric pseudoflow solver is based on a completely different implementation, there was no problem with the parametric pseudoflow solver.

Therefore, we opted to model each block with two arcs, (s, v) and (v, t); however, only one of the two arcs ever has positive capacity. This increases the number of arcs in the graph from m to $m + n$. These added arcs make updating the graph for a new value of λ much more efficient, although processing these additional arcs (even though they have zero capacity) does result in a performance penalty. This penalty is clearly evident when solving for a single parameter (e.g., Table 4.3). However, given even a modest number of parameter values, the improved efficiency of the parametric push-relabel solver far outweighs the added overhead of the additional arcs.

4.7.2 Image Segmentation

We tested with two image data files, one had 16,384 nodes, and the other had 48,092 nodes. Both images had 4 bits per pixel or 16 possible grayscale values. We tested with 14 parameter values from 1 to 14. The heuristic combination for the pseudoflow solvers used simple initialization, constant initial node labels, pre-order searches, LIFO branch management, and no global relabeling (simple-const-highest/lowest-pre-lifo-0).

Table 4.4 shows the execution times (in seconds) for the solvers running with image data. For each iterative/parametric pair, the best time is shown in bold font followed by the ratio of the iterative time to the parametric time (i.e., a higher ratio indicates better performance for the parametric solver).

	image1	image2
num pixels	16,384	48,090
iter-hl	1.55	6.49
param-hl	**0.57**	**2.98**
ratio	2.72	2.18
iter-pfs-lo	2.55	7.27
param-pfs-lo	**0.61**	**2.04**
ratio	4.18	3.56
iter-pfs-hi	2.19	9.61
param-pfs-hi	**0.92**	**5.19**
ratio	2.38	1.85

Table 4.4: Running times for parametric and iterative solvers for image data with 14 parameter values: $\lambda \in \{1 \dots 14\}$.

We can see that again the parametric solvers run considerably faster than the iterative

solvers. The lowest label parametric pseudoflow solver achieves the highest speed-up ratios and is the fastest solver for the larger instance, *image2*. The parametric push-relabel is the fastest for the smaller instance, *image1*.

Table 4.5 shows the details of solving times, number of pushes, and number of mergers (for the pseudoflow algorithm) for each value of λ for the smaller file, *image1*.

Again, as expected, the operation counts for subsequent values of λ when using any of the parametric solvers are are generally substantially less than solving the problems from scratch. For this instance, the parametric solvers perform between 165% (115,563 versus 43,541 for lowest label pseudoflow) and 270% (276,472 versus 75,349 for highest label pseudoflow) more push operations and take from 80% (0.56 sec versus 0.31 sec for lowest label pseudoflow) to 207% (0.86 sec versus 0.28 sec for highest label pseudoflow) more time to solve the entire sequence than solving the most difficult single instance. So, all of the solvers were able to solve a sequence of 14 problems with less than three times the work needed to solve the single worst instance.

Solver	λ Value 1	2	3	4	5	6	7	8	9	10	11	12	13	14	Total
	Cut Time														
iter-hl	0.01	0.02	0.03	0.06	0.14	0.12	0.11	0.11	0.13	0.15	0.15	0.16	0.16	0.20	1.55
param-hl	0.05	0.04	0.06	0.06	0.08	0.06	0.04	0.03	0.03	0.02	0.03	0.02	0.03	0.02	0.57
ratio	0.20	0.50	0.50	1.00	1.75	2.00	2.75	3.67	4.33	7.50	5.00	8.00	5.33	10.00	2.72
iter-pfs-lo	0.02	0.03	0.08	0.14	0.22	0.27	0.31	0.28	0.23	0.21	0.22	0.21	0.22	0.11	2.55
param-pfs-lo	0.02	0.00	0.06	0.10	0.15	0.09	0.04	0.02	0.04	0.01	0.01	0.01	0.01	0.00	0.56
ratio	1.00		1.33	1.40	1.47	3.00	7.75	14.00	5.75	21.00	22.00	21.00	22.00		4.55
iter-pfs-hi	0.02	0.03	0.07	0.14	0.21	0.28	0.21	0.19	0.22	0.20	0.16	0.16	0.15	0.15	2.19
param-pfs-hi	0.02	0.02	0.06	0.10	0.26	0.17	0.09	0.04	0.05	0.01	0.01	0.01	0.01	0.01	0.86
ratio	1.00	1.50	1.17	1.40	0.81	1.65	2.33	4.75	4.40	20.00	16.00	16.00	15.00	15.00	2.55
	Number of Pushes														
iter-hl	337	4,920	19,156	31,884	63,397	62,685	53,175	60,954	50,634	48,466	48,380	49,127	48,839	49,858	591,812
param-hl	217	3,131	20,284	28,005	49,048	36,109	12,688	7,363	9,413	1,024	1,059	1,114	1,523	1,523	172,501
ratio	1.55	1.57	0.94	1.14	1.29	1.74	4.19	8.28	5.38	47.33	45.68	44.10	32.07	32.74	3.43
iter-pfs-lo	257	3,789	19,326	26,952	43,541	35,177	20,004	13,382	9,289	3,743	3,119	2,706	2,859	1,098	185,242
param-pfs-lo	257	3,169	15,698	19,499	35,935	19,467	7,002	5,400	5,361	805	804	882	1,043	241	115,563
ratio	1.00	1.20	1.23	1.38	1.21	1.81	2.86	2.48	1.73	4.65	3.88	3.07	2.74	4.56	1.60
iter-pfs-hi	254	3,685	19,754	39,517	68,203	75,349	44,174	39,550	39,406	34,469	33,211	33,576	34,257	33,755	499,160
param-pfs-hi	254	3,972	22,111	35,899	96,655	60,779	23,920	9,880	14,790	2,489	1,312	1,002	1,886	1,523	276,472
ratio	1.00	0.93	0.89	1.10	0.71	1.24	1.85	4.00	2.66	13.85	25.31	33.51	18.16	22.16	1.81
	Number of Mergers														
iter-pfs-lo	220	2,372	11,353	16,078	24,954	20,372	12,052	8,245	5,699	2,509	2,087	1,772	1,789	724	110,226
param-pfs-lo	220	2,007	9,102	11,352	20,789	10,814	3,915	3,235	3,015	470	481	523	594	88	66,605
ratio	1.00	1.18	1.25	1.42	1.20	1.88	3.08	2.55	1.89	5.34	4.34	3.39	3.01	8.23	1.65
iter-pfs-hi	244	2,760	13,883	28,966	51,743	59,576	39,043	36,024	36,437	33,447	32,595	32,947	33,484	33,258	434,407
param-pfs-hi	244	2,846	14,511	23,718	64,168	42,694	16,822	6,917	10,433	1,815	887	713	1,279	1,060	188,107
ratio	1.00	0.97	0.96	1.22	0.81	1.40	2.32	5.21	3.49	18.43	36.75	46.21	26.18	31.38	2.31

Table 4.5: Cut time, number of push operations and merger operations for parametric and iterative, push-relabel and pseudoflow solvers for image data file *image1*.

4.7.3 RLG Extra Wide

We tested with three files from the RLG extra wide class. The files, *rlg.16*, *rlg.17*, and *rlg.18*, had approximately 2^{16}, 2^{17}, and 2^{18} nodes, respectively. We tested with seven values of λ: 50, 75, 100, 125, 150, 175, and 200. The pseudoflow solvers used simple initialization with constant initial labels, wave branch management, pre-order searching, and no global relabeling (simple-const-highest/lowest-wave-pre-0).

Table 4.6 shows the execution times (in seconds) for the solvers running with RLG extra wide data. For each iterative/parametric pair, the best time is shown in bold font followed by the ratio of the iterative time to the parametric time (i.e., a higher ratio indicates better performance for the parametric solver).

	rlg.16	rlg.17	rlg.18
nodes	65,538	131,074	262,146
arcs	180,224	360,448	720,896
iter-hl	1.39	4.85	11.25
param-hl	**0.64**	**1.89**	**4.31**
ratio	2.17	2.57	2.61
iter-pfs-lo	6.58	19.76	41.78
param-pfs-lo	**1.53**	**3.90**	**8.27**
ratio	4.30	5.07	5.05
iter-pfs-hi	3.40	10.47	21.71
param-pfs-hi	**0.85**	**2.00**	**4.30**
ratio	4.00	5.24	5.05

Table 4.6: Running times for parametric and iterative solvers for RLG extra wide data with $\lambda \in \{50, 75, 100, 125, 150, 175, 200\}$.

As was the case previously, the parametric solvers run faster than the iterative solvers. The parametric pseudoflow solvers consistently achieve a higher speedup-ratio than the parametric push-relabel solver. However, in all cases, the parametric push-relabel solver is faster in absolute terms compared to the parametric pseudoflow solvers.

Table 4.7 shows the details of solving times, number of pushes, and number of mergers (for the pseudoflow algorithm) for each value of λ for the *rlg.17* file.

Solver	λ Value 50	75	100	125	150	175	200	Total
	Cut Times							
iter-hl	0.87	0.80	0.76	0.71	0.62	0.56	0.53	4.85
param-hl	1.01	0.18	0.16	0.14	0.14	0.13	0.12	1.88
ratio	0.86	4.44	4.75	5.07	4.43	4.31	4.42	2.58
iter-pfs-lo	3.30	2.93	3.00	2.94	2.52	2.52	2.55	19.76
param-pfs-lo	3.27	0.23	0.09	0.08	0.07	0.06	0.05	3.68
ratio	1.01	12.74	33.33	36.75	36.00	42.00	51.00	5.37
iter-pfs-hi	1.57	1.54	1.49	1.50	1.48	1.47	1.42	10.47
param-pfs-hi	1.56	0.09	0.07	0.06	0.05	0.05	0.05	1.76
ratio	1.01	17.11	21.29	25.00	29.60	29.40	28.40	5.95
	Number of Pushes							
iter-hl	520,559	433,319	377,433	333,817	304,131	277,067	253,915	2,500,241
param-hl	520,559	16,834	8,960	3,231	2,097	1,172	321	553,174
ratio	1.00	25.74	42.12	103.32	145.03	236.41	791.01	4.52
iter-pfs-lo	347,562	266,878	212,822	176,490	150,318	131,348	118,001	1,403,419
param-pfs-lo	347,562	4,673	2,249	1,088	606	341	178	356,697
ratio	1.00	57.11	94.63	162.22	248.05	385.18	662.93	3.93
iter-pfs-hi	322,600	292,025	273,855	260,575	251,481	244,117	237,880	1,882,533
param-pfs-hi	322,600	5,194	2,404	1,127	670	374	194	332,563
ratio	1.00	56.22	113.92	231.21	375.34	652.72	1226.19	5.66
	Number of Mergers							
iter-pfs-lo	181,978	153,622	133,278	118,297	106,941	98,242	91,669	884,027
param-pfs-lo	181,978	949	437	200	125	59	34	183,782
ratio	1.00	161.88	304.98	591.49	855.53	1665.12	2696.15	4.81
iter-pfs-hi	274,498	256,810	244,917	235,705	228,565	222,929	217,990	1,681,414
param-pfs-hi	274,498	1,170	505	235	153	70	38	276,669
ratio	1.00	219.50	484.98	1003.00	1493.89	3184.70	5736.58	6.08

Table 4.7: Cut times, number of pushes and mergers for parametric and iterative pseudoflow solvers for RLG extra wide data file *rlg.17*.

As expected, the operation counts for subsequent values of λ when using any of the parametric solvers are are generally substantially less than solving the problems from scratch. For all of the parametric solvers, the number of pushes to solve the entire sequence is less than 5% more than is required to solve the most difficult single instance. The parametric pseudoflow lowest label solver takes only 12% more time (3.68 sec versus 3.30 sec) to solve the whole sequence than the hardest single instance, the parametric pseudoflow highest label solver also takes 12% more time (1.76 sec versus 1.57 sec), and the parametric push-relabel solver takes approximately twice as long (1.88 sec versus 0.87 sec) as the single hardest instance.

4.8 Conclusions

In this chapter, we describe a specific type of parametric maximum flow/minimum cut problem and provide some sample applications from the literature. We describe how these problems can be solved efficiently using either the push-relabel or the pseudoflow algorithms. Both of these algorithms can perform simple parametric analysis on at least $O(n)$ parameter values with the same complexity as a single run of the corresponding nonparametric maximum flow algorithm.

Our experimental results show that these algorithms are also efficient in practice. We find that the parametric implementations do, in fact, run faster than solving the problems iteratively. In some cases, we can solve a sequence of approximately 10 parametric problems with less than 20% more effort than is required to solve the single most difficult instance. Thus, these parametric algorithms could be very useful in practice.

Compared against each other, the results of the parametric versions of the push-relabel and pseudoflow algorithms are mixed, with neither being consistently better than the other.

Chapter 5

Warm Start

In this chapter, we discuss a warm start technique for the pseudoflow algorithm. Like the parametric algorithm of the previous chapter, warm start allows us to use the results of one solution to create an initial normalized forest to solve a subsequent problem. Unlike the parametric networks in Chapter 4, the warm start algorithm allows the capacities of any arc to be changed, not just the source and sink arcs.

5.1 Introduction

Given an initial, underlying network $G = (V, A)$, we can define a sequence of networks, G_1, G_2, ..., G_k, where the nodes and arcs of each network are the same, but each network G_i has a different set of capacities on the arcs, c_i. The capacities may differ in any arbitrary way, but the capacities must still be non-negative. Our objective is to compute the minimum cut for each network. These instances may be generated dynamically—i.e., the entire sequence need not be known when we begin the algorithm.

This technique was originally motivated by the scheduling algorithm of Möhring et al. [MSSU03] that we described briefly in Section 3.4.2.7. The authors solve the scheduling problems by solving a series of minimum cut instances. These instances have no parametric functions relating the capacities in one instance to those in another. The authors solve each instance independently using the push-relabel algorithm.

5.2 Warm-Start Pseudoflow Algorithm

The warm start algorithm uses the results from solving one instance as initial data for solving the next instance. If the solution to the next instance is expected to be similar to the solution for the previous instance, it seems likely that the previous solution would be a good initial starting point for solving the next instance.

After completing Phase I for an instance G_i and computing pseudoflow values f_i, we read in the new capacities c_{i+1} for instance G_{i+1} and adjust the flows to create an initial pseudoflow f_{i+1} to begin solving G_{i+1}.

There are a large number of heuristics that can be used when adjusting the flows on the arcs. The only requirement is that we meet the requirements of the normalized forest structure described in Section 2.1.2. We present a simple heuristic below and will discuss other heuristics later.

Obviously, if the new capacity of an arc is less than the previous flow on the arc, $c_{i+1}(u,v) < f_i(u,v)$, we need to reduce the flow to produce a feasible pseudoflow. We set the flow to match the new capacity—i.e., $f_{i+1}(u,v) = c_{i+1}(u,v)$.

If the capacity of an arc increases, the existing flow on the arc would be feasible and does not necessarily require adjusting. However, the normalized forest structure requires that all out-of-tree arcs are at either their upper capacity bound or zero. Therefore, if the arc was saturated before, $f_i(u,v) = c_i(u,v)$, then we choose to keep it saturated by setting the new flow to match the new capacity, $f_{i+1}(u,v) = c_{i+1}(u,v)$.

procedure adjustFlows($A, f_i, f_{i+1}, c_i, c_{i+1}$)**:**
  foreach arc $(u,v) \in A$**:**
    if $c_{i+1}(u,v) < f_i(u,v)$**:**
      $f_{i+1}(u,v) \leftarrow c_{i+1}(u,v)$
    else if $c_i(u,v) = f_i(u,v)$:
      $f_{i+1}(u,v) \leftarrow c_{i+1}(u,v)$
    else:
      $f_{i+1}(u,v) \leftarrow f_i(u,v)$

Clearly, if we adjust the flow on an arc, we will generate excess on one end and deficit on the other. This may violate the normalized forest property that requires non-root nodes have zero excess. Therefore, we renormalize the branches in a way that is similar to that

used for the parametric pseudoflow algorithm in Section 4.4. One difference is that the parametric algorithm only deals with excess at non-root nodes adjacent to the source or sink, whereas the warm start algorithm must deal with excess or deficit anywhere in the graph—i.e., the renormalization procedure for the warm start algorithm is more general than the renomalization proceure for the parametric algorithm.

We use a post-order traversal [CLR90] to process the nodes within a branch from the leaves up to the root. If a node has positive excess, we attempt to push the excess up to the node's parent. If the node has deficit, we attempt to pull excess down from the parent. If the edge between the node and its parent does not have sufficient residual capacity to move the entire amount of excess, the node is split from its parent, and the node becomes the root of a new branch.

```
procedure renormalize(V_I, f, c, par):
  foreach branch root v ∈ V_I:
    renormalizeSubTree(v, f, par)

procedure renormalizeSubTree(v, f, c, par):
  foreach child u of v:
    renormalizeSubTree(u, f, c, par)
  if e_f(v) ≠ 0:
    p ← par(v)
    if e_f(v) > 0:
      δ ← min(e_f(v), r_f(v, p))
    else:
      δ ← max(e_f(v), −r_f(p, v))        ⟨⟨δ is negative⟩⟩
    f(v, p) ← f(v, p)+δ
    if e_f(v) ≠ 0:
      ⟨⟨split v from p⟩⟩
      par(v) ← nil
      if e_f(v) > 0:
        mark v strong
      else:
        mark v weak
```

Hochbaum [Hoc97] showed that this renormalization procedure runs in time $O(n)$ because each in-tree arc is traversed and updated $O(1)$ times.

For the highest and lowest label algorithms, we may also need to adjust the labels of the nodes. As a result of adjusting the capacities and flows as well as the renormalization

procedure, the node labels may no longer represent lower bounds on the distance from the nodes to the sink in the residual graph , thus violating Property 3 for the node labels (see Section 2.3).

There are numerous possible policies for adjusting the labels. We choose to relabel the nodes using the constant labeling scheme defined in Section 2.3: weak nodes are assigned label one and strong nodes are assigned label two. An obvious alternative would be a distance-to-deficit labeling described in Section 2.6.3. We found this policy performed so poorly during our initial testing that we did not thoroughly test it, and it is not used in our experiments reported later in the chapter.

The pseudocode for the entire warm start algorithm is shown below.

procedure warmStartPseudoflow($V, A, s, t, c_1, \ldots, c_k$)**:**
  read first instance capacities c_1
  call pseudoflowPhase1(V, A, s, t, c_1, f_1)
  for $i = 2$ **to** k**:**
    read capacities c_i
    call adjustFlows($A, f_{i-1}, f_i, c_{i-1}, c_i$)
    call renormalize(V_I, f_i, c_i)
    $\langle\langle$*adjust the labels*$\rangle\rangle$
    foreach $v \in V_I$**:**
      if v is strong**:**
        $l(v) \leftarrow 2$
      else:
        $l(v) \leftarrow 1$
    call pseudoflowPhase1(V, A, s, t, c_i, f_i)

The complexity results for both the standard pseudoflow algorithm and the parametric pseudoflow solver rely on the maximum value of the node labels being less than n and the property that the labels never decrease. These two conditions allow the number of mergers to be bounded.

Because the capacities of any or all arcs in the graph may change when moving from one warm start instance to the next, a node may become much closer to the the sink in the new residual graph. Property 3 for the node labels requires that the label of a node be a lower bound on the distance from the node to the sink in the residual graph. So, in order to preserve Property 3, the label of the node must be reduced.

Thus, we cannot bound the number of meregers, which means we cannot provide any complexity bounds beyond that of repeatedly solving each instance independently. In the absence of any special structure that the algorithm could exploit, this is what we would expect: to solve k instances, the complexity is k times that of a single run.

5.2.1 Heuristics for Adjusting Flows

In the previous section, we presented a simple policy for adjusting the flows on arcs in response to new capacities. For saturated, out-of-tree arcs, when the capacity of the arc increases, the policy always keeps the arc saturated. However, other policies are possible for dealing with saturated, out-of-tree arcs that increase capacity, so long as the result is a normalized forest. In particular, out-of-tree arcs must be either saturated or have zero flow in accordance with property 2 of normalized forests (see Section 2.1.2). Note that for our experiments later in the chapter, we only use the simple policy.

We call one such policy "rounding." Let δ be the amount by which the capacity of arc (u, v) increases: $\delta = c_{i+1}(u, v) - c_i(u, v)$. The simplest rounding rule keeps the arc saturated if the change in capacity is less than the previous capacity (if $\delta < c_i(u, v)$ then $f_{i+1}(u, v) = c_{i+1}(u, v)$). Otherwise, we reduce the flow on (u, v) to zero ($f_{i+1}(u, v) = 0$). The idea is to "round" the flow up or down, depending on which requires a bigger change in the flow.

We can further enhance this policy by introducing a parameter $\alpha > 0$ such that if $\alpha\delta < c_i(u, v)$, then we round the flow up, $f_{i+1}(u, v) = c_{i+1}(u, v)$. Otherwise, we reduce the flow on (u, v) to zero. If $\alpha = 1$, this is the same as the simple rounding rule above. Otherwise, if $\alpha < 1$, then more arcs will be kept saturated (rounded up), and if $\alpha > 1$, then more arcs will be reduced to zero (rounded down).

Both of these procedures ensures that out-of-tree arcs are either saturated or have zero flow. Regardless of the procedure used to adjust the flows on arcs, the same renormalization procedure (*renormalize*) can be used to renormalize the branches after the adjustment to produce a valid normalized forest.

5.3 Software

We extended the pseudoflow solver by adding functions to read a new set of capacities, adjust the flows, and renormalize the branches. The code to adjust the flows is based on the simple *adjustFlows* procedure in the previous section, which does not perform rounding. We wrote a new driver program that calls these functions for each problem instance and collects the time to establish a cut—i.e., solve the instance.

We modified the workbench to call the new warm start driver program. We also created a new iterative solver that is similar, but simpler than the one we used in Chapter 4. The iterative solver uses a a simple, non-warm-start maximum flow solver (either pseudoflow or push-relabel from Chapter 3) to solve a sequence of minimum cut problems. This is simpler than the iterative solver from Chapter 4 in that this solver does not need to substitute parameter values into a parametric graph to create non-parametric instances. Rather, we are given a sequence of non-parametric instance to solve, and we solve them in the order given.

5.4 Hardware

The experiments in this section were performed on a different Sun Microsystems computer than that described in Section 3.3 (and also used in Chapter 4). The performance results presented here were conducted on a Sun E250 system. The system had one Sparc V9 CPU running at 400 MHz with 640 MB of RAM.

We again performed the machine calibration experiment, as suggested by the DIMACS Challenge Core Experiments [Dim90]. Table 5.1 shows the running times (in seconds) for the two tests with and without compiler optimizations.

Optimization	Test 1			Test 2		
Level	real	user	system	real	user	system
no optimization	0.2	0.2	0.0	1.9	1.9	0.0
-O flag	0.1	0.1	0.0	1.2	1.2	0.0

Table 5.1: Average running times for DIMACS machine calibration tests on Sun E250 system.

5.5 Testing Methodology

5.5.1 Testing Procedure

We wish to evaluate how well the warm start technique works compared to solving the sequence of problems iteratively. For each problem instance, we compare the running times of the warm start solver to that of the iterative solver and report the total time to establish the k minimum cuts, where k is specific to each problem instance.

We average the times over five runs for each solver-instance pair. For the warm start solver, the time includes establishing the initial normalized forest, solving the k problems, and renormalizing the graph $k-1$ times between solutions. For the iterative solver, the time is the sum of the times to solve the k instances including establishing the initial normalized forest and solving the instances. Both solvers exclude the time spent reading the instances into memory. All times represent user CPU time in seconds as reported by the POSIX `times` system call [Pos88].

For both solvers, we use the best heuristic combinations we identified in Chapter 3 for each problem class. These heuristics are specified with the results for each problem class.

5.5.2 Data Files

Our first set of test instances come from the original scheduling problems of Möhring et al. [MSSU03] and is described in Section 3.4.2.7. We selected a larger number of instances than we used in Chapter 3. The problems vary in size and difficulty (from 4,200 to 20,500 nodes - see Table 5.2). Each scheduling instance contains 51 minimum cut problems.

We also developed a problem generator that converts a single network into a sequence of instances where each instance differs from the previous one in that the arc capacities are adjusted as described below. This generator was applied to random level graphs (RLG) as described in Chapter 3.

The generator is controlled by two parameters, a and c, that take values between zero and one. For each arc in the graph, we flip a biased coin to decide if the arc capacity will change. That is, we select a random number q from $U[0,1]$, and if q is less than or equal to a, we adjust the arc. To adjust the arc capacity, we select a second random number r from $U[-c,c]$ and set $c(u,v) = (1+r)c(u,v)$. As we generate a sequence of

instances, each instance is the input for the next instance—i.e., each instance is expected to be increasingly different from the original instance.

We also developed a variant of this generator for use with mining graphs. The generator takes a mining instance and changes the source and sink arcs representing the ore content of the blocks in the mine to create a new mining instance.

These changes differ from the type of parametric analysis performed in the previous chapter where we simulate changes in the price of the ore, given a fixed estimate of the amount of ore in the blocks. For example, if the price of the ore increases, the economic value of all blocks continaing ore will increase. Changing the values of the blocks arbitrarily allows one to simulate changes in the amount of ore thought to be in the blocks, and the changes need not vary according to a monotone function or a single parameter. Hence, the economic value of a block could increase, decrease, or remain unchanged.

5.6 Results

5.6.1 Scheduling Data

The running times for scheduling problem instances are shown in Table 5.2. The heuristics used are simple-const-highest-lifo-pre-0. We can see that the warm start technique is faster (ratio greater than one) in four of the instances (j1, j22, j28, and j38), but slower in three other instances (j6, j16, and j56). For the remaining two instances (j7 and j27), the performance for warm start is the same as iterative. In the best case (j22), the performance is only 1.7 times faster than iterative.

	j1	j6	j7	j16	j22	j27	j28	j38	j56
nodes	5,194	11,617	5,770	18,395	4,257	6,143	4,745	4,523	20,565
arcs	11,068	27,487	12,820	44,416	9,966	15,258	10,901	10,185	60,567
warm-pfs	1.398	7.246	2.704	15.698	1.492	4.040	2.214	1.774	21.612
iter-pfs	1.858	6.186	2.660	11.086	2.522	4.048	2.632	2.140	15.630
ratio	**1.33**	0.85	0.98	0.71	**1.69**	1.00	**1.19**	**1.21**	0.72

Table 5.2: Running times for warm start and iterative solvers for scheduling data. Ratios greater than one (shown in bold) indicate that the warm start solver performs better than the iterative solver.

Graph Size

Notice that warm start performs better than the iterative solver for smaller graphs (between 4,200 and 5,200 nodes) and worse for larger graphs (over 25,000 nodes). The performance is approximately equal for the graphs containing 5,800 nodes and 6,200 nodes.

Recall from Section 3.4.2.7 that the scheduling minimum cut problem instances are composed of J paths, one for each job being scheduled, and each path is approximately T nodes long, one node for each time period. Each instance contains 120 jobs. In other words, the graphs are all 120 rows wide, but the number of time periods, the length of the graph or number of columns, varies. So, the warm start algorithm seems to work better for shorter scheduling graphs. Intuitively, the length of the graph should not hinder the warm start technique. If longer graphs are problematic, we would expect that they would also be problematic for the iterative solver.

Number of Arc Changes

Next, we looked at the number of arcs that changed capacity and the magnitude of the changes between consecutive instances. There are two types of arcs in the scheduling graphs. The arcs along the paths corresponding to jobs have finite capacities that vary between instances in a sequence. The arcs between paths that represent dependencies between the jobs have infinite capacity, and the capacities for these arcs do not change. Therefore, in the discussions that follow, we are only concerned with the finite capacity arcs along the paths.

Table 5.3 shows the performance of warm start (as a speedup ratio) and the average percentage number of the finite arcs that changed capacity between the subsequent instances in the scheduling problems. We can see that for instances in which warm start performs relatively poorly, nearly all of the arcs (over 95%) change. In the instances where warm start performs well, fewer arcs change, but the percentage is still fairly large—65% to 87%.

Instance	j1	j6	j7	j16	j22	j27	j28	j38	j56
Ratio	**1.33**	0.85	0.98	0.71	**1.69**	1.00	**1.19**	**1.21**	0.72
%arcs changed	64.61%	94.79%	95.94%	96.65%	66.54%	93.97%	85.21%	86.88%	96.60%

Table 5.3: Average percentage of finite capacity arcs changing capacity and speedup ratio for warm start.

Magnitude of Capacity Changes

The magnitude of the changes of the arc capacities follows the same pattern for all of the scheduling instances. The initial instances in the sequences show tremendous percentage change in capacity, but the changes taper off towards the end of the sequence, where the capacities changed by only 1–3%. This is shown in Figure 5.1 along with the speedup ratio for each instance in the sequence. The top two instances (j6 and j56) are ones where warm start does poorly, and the bottom two instances (j22 and j28) are instances where it does well.

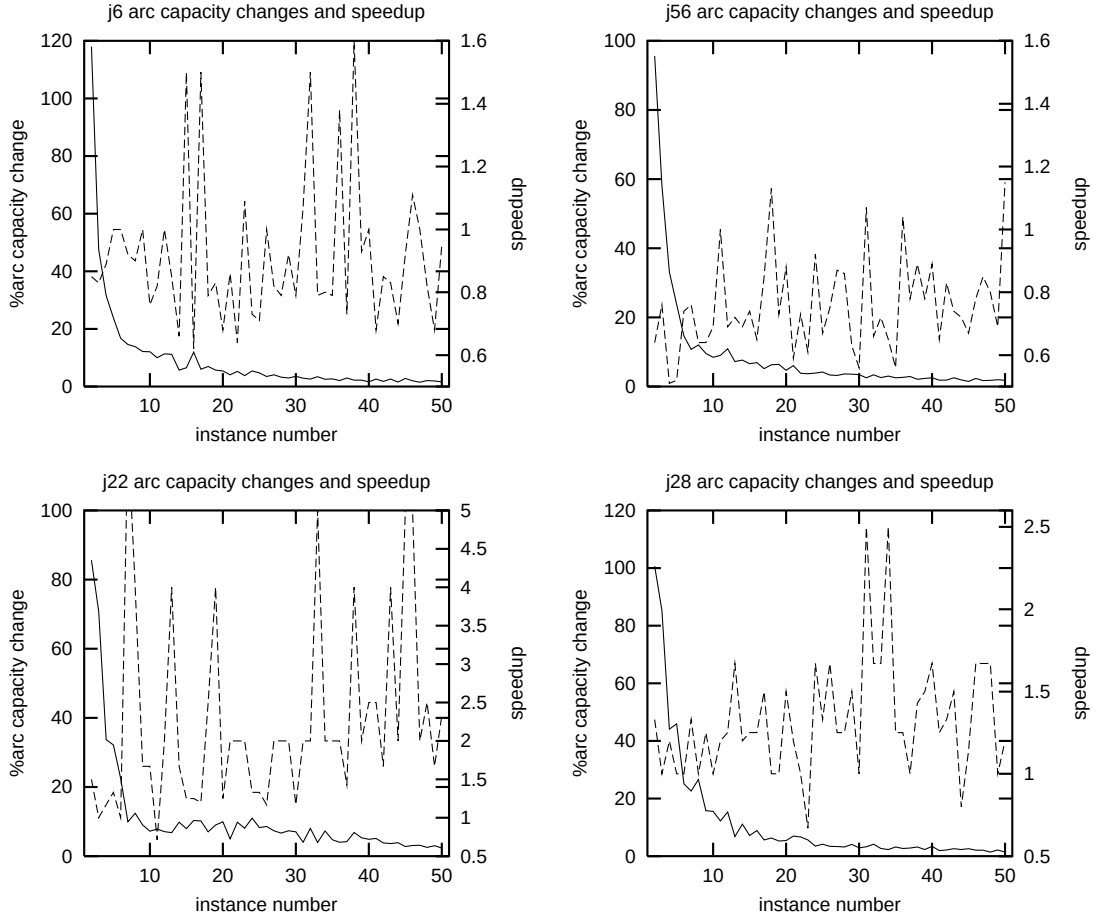


Figure 5.1: Plots of the magnitude of capacity changes and speedup ratio through the problem sequences for a subset of the scheduling instances. The capacity changes are plotted on the left-hand side with the solid line. The speedup is plotted on the right-hand scale with the dashed line.

In all cases, the arc changes follow the same pattern regardless of whether warm start

performs better or worse than iterative solver. There seems to be no correlation between the speedup ratio and the arc capacity changes or the instance number.

Changes in Number of Strong Nodes

In an attempt to quantify how much work the warm start algorithm does when solving, we looked at the change in the number of strong nodes before solving an instance (after renormalization) and the number after solving it. This crude estimate looks only at the number of strong nodes, not the specific nodes. This estimate will not be representative in cases where the solver performs a lot of work, but finishes with the same number of strong nodes as it began with.

We plotted the percentage change in the number of strong nodes against the speedup ratio for all of the scheduling instances in Figure 5.2 below. The plot has a bell-curve shape, centered to the right of zero.[1] It appears that the warm start solver may do poorly regardless of the change in the number of strong nodes. In those cases where the warm start solver performs well, the number of strong nodes either decreased slightly or increased. However, in those same cases, there are many examples where the warm start solver performed poorly. So, the warm start solver may do well when the change in the number of strong nodes is modest, but a modest change does not guarantee good performance.

Renormalization Overhead

Another possible explanation for the poor performance of warm start is that the cost of renormalizing the branches for the warm start solver exceeds the cost of initializing the normalized forest from scratch for the iterative solver.

We looked at the cumulative amount of time spent establishing the normalized forest before solving each instance. For the iterative solver, this is the sum of the initialization times for each instance. For the warm start solver, it is the first initialization time plus the sum of all subsequent renormalization times. We call these times the "tree times" for each algorithm.

Table 5.4 below shows the overall solving time, the tree time, and the time spent solving (excluding the tree time) for the iterative and warm start solvers for four scheduling

[1]The data above 1.5 occurs in bands due to timer resolution issues. The resolution of the measurements is 0.01 seconds, and we are comparing values like 0.01 vs. 0.02 and 0.02 vs. 0.05.

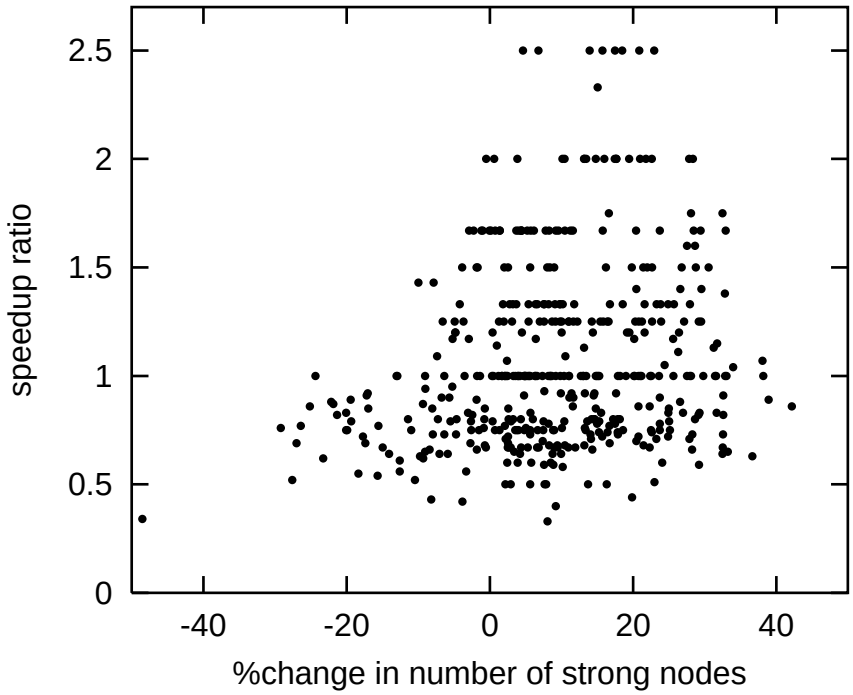


Figure 5.2: Plot of the speedup ratio as a function of the change in the number of strong nodes while solving for all of the scheduling instances. The y-axis was truncated at 2.5 to better show the details of the rest of the data.

problems: two where warm start performs well and two where it does not. We also include the ratio of the iterative times to the warm start times.

We use the simple initialization scheme for solving scheduling instances. For all of the instances, renormalization is more expensive than repeated initialization. This is probably because renormalization makes two passes over the nodes (once to push excess or deficit to the roots and once to reset the node labels). However, as we can see, the extra tree time for the warm start solver does not entirely explain the poor performance of the warm start solvers; the solving times (excluding the tree times) are still slower than the iterative solver for j16 and j56. So, renormalization results in a normalized forest that is "farther" from optimal compared with the initial normalized forest. In other words, it requires more effort to solve starting with a renormalized forest than to solve with an initial normalized forest.

Operations Counts

Solver	Time	j16	j56	j22	j28
Warm	Overall	15.88	21.95	1.70	2.06
	Tree	2.37	3.04	0.32	0.31
	Solve	13.51	18.91	1.38	1.75
Iterative	Overall	11.03	15.9	2.61	2.63
	Tree	0.64	0.87	0.13	0.09
	Solve	10.39	15.03	2.48	2.54
Ratio	Overall	0.69	0.72	**1.54**	**1.28**
	Tree	0.27	0.29	0.41	0.29
	Solve	0.77	0.79	**1.80**	**1.45**

Table 5.4: Tree times: comparing renormalization in the warm start solver to re-initialization in the iterative solver.

In Table 5.5 we present data regarding the relative number of operations performed by the warm start solver compared to the iterative solver. Each entry is the ratio of the number of operations performed by the iterative solver compared to the warm start solver. Higher ratios indicate that the warm start solver is performing fewer operations, which should yield better performance.

	j6	j16	j56	j22	j28	j38
times	0.88	0.71	0.70	**1.80**	**1.23**	**1.27**
mergers	1.95	1.71	1.47	3.12	1.90	2.17
pushes	1.20	0.94	0.91	2.71	1.29	1.26
arc scans	1.03	0.82	0.85	2.20	1.49	1.54
node visits	0.95	0.74	0.75	2.09	1.36	1.38
rehangs	0.43	0.19	0.21	1.77	0.90	0.46
relabels	0.99	0.81	0.85	2.00	1.42	1.42
splits	0.55	0.24	0.21	2.13	1.12	0.70

Table 5.5: Operation count ratios for scheduling instances. Each entry represents the ratio of the number of operations performed by the iterative solver vs. the number performed by the warm start solver. Values greater than one indicate warm start is performing fewer operations.

We can see that the warm start solver always performs fewer merger operations. However, each merger operation is composed of a number of other operations, and for these, the data follows a pattern: in cases where warm start does well, it performs fewer of these operations (ratio greater than one), and in the others it performs more.

A greater number of node visits and arc scans indicates that warm start performs more work to identify a merger arc. The higher number of pushes and rehangs indicates that the path between the strong root and the weak root is longer, and that the branches are bigger. The higher number of rehangs indicates that the warm start solver is identifying merger arcs deeper in the strong branch. So, despite the fact that warm start always performs fewer mergers, when the warm start algorithm does poorly, it performs more work to identify the merger arc and to renormalize the tree after the merger.

Summary

The warm start technique shows mixed results with the scheduling data. It runs faster for smaller, shorter graphs than the iterative solver. It seems to run faster when fewer arcs are changed. When it runs slower, the warm start algorithm still performs fewer mergers, but it does more work to identify and execute the mergers.

5.6.2 Random Level Graphs—RLG

To gain more experimental data and to test some of the hypotheses regarding the performance from the previous section, we generated sequences of instances based on random level graphs (RLG).

Using the generator described in Section 5.5.2, we generated 16 *rlg-long* instances with 2^{15} (32,768) nodes. We varied the percentage of arcs changed ($a = 20, 80$) and the magnitude of the changes ($c = 20, 80$). For each combination of a and c, we generated four instances each with a different random number seed. Each instance is a sequence of 10 problems.

The results of the warm start solver on the *rlg-long.15* instances are shown in Table 5.6. The heuristic combination used was simple-deficit-highest-wave-pre-0.

Overall, the warm start technique performs better on *rlg-long* than on the scheduling data. There are a number of cases where the speedup ratio exceeds 2.0 and the highest is 3.33 (a=80, c=80, seed=30). On the other hand, the lowest speed up ratio is 0.4 (a=80, c=20, seed=30), which is worse than any of the scheduling instances.

Intuitively, we would expect that smaller magnitude changes affecting a smaller number of arcs (a20c20) would yield the best performance because the source sets of the cuts

	a20c20			a20c80			a80c20			a80c80		
seed	warm	iter	ratio	warm	iter	ratio	warm	iter	ratio	warm	iter	ratio
10	7.754	8.828	**1.14**	11.942	20.292	**1.70**	8.122	12.256	**1.51**	12.012	23.576	**1.96**
20	7.414	8.494	**1.15**	7.246	5.336	0.74	9.142	10.078	**1.10**	13.316	29.022	**2.18**
30	7.198	7.460	**1.04**	8.330	7.400	0.89	10.636	4.232	0.40	14.856	49.450	**3.33**
40	7.128	6.442	0.90	11.498	31.842	**2.77**	8.136	5.316	0.65	15.736	31.912	**2.03**
total	29.494	31.224	**1.06**	39.016	64.870	**1.66**	36.036	31.882	0.88	55.920	133.960	**2.40**

Table 5.6: *Rlg-long* warm start performance. The values for a (20, 80) indicate the percentage of arcs whose capacity was changed, and the values for c (20,80) indicate the percentage change in capacity. The heuristic combination used was simple-deficit-highest-wave-pre-0.

between successive instances should tend to change less. At the other extreme, we would expect that large magnitude changes on a large number of arcs (a80c80) would yield the worst performance. However, we can see no such pattern.

Graph Length

One theory from the scheduling data is that warm start performs better on shorter graphs, possibly due to smaller branches. To test this hypothesis, we generated some *rlg-xwide* instances similar to those used in Chapter 4. These graphs have the same number of nodes as the *rlg-long* graphs above, but only have four columns of nodes between the source and the sink (vs. 512 columns for *rlg-long*).

The results of the *rlg-xwide* data are shown below in Table 5.7. There is little difference in the effectiveness of the warm start technique between the longer *rlg-long* graphs and the shorter *rlg-xwide* graphs; warm start is still not uniformly better than iterative. The performance of warm start does seem more robust: the range of speedup ratios for *rlg-xwide* graphs is considerably smaller—0.87 to 1.41 vs. 0.4 to 3.33 for *rlg-long*.

	a20c20			a20c80			a80c20			a80c80		
seed	warm	iter	ratio	warm	iter	ratio	warm	iter	ratio	warm	iter	ratio
10	2.23	3.14	**1.41**	3.12	3.24	**1.04**	2.77	3.37	**1.22**	3.94	3.52	0.89
20	2.27	3.14	**1.39**	3.16	3.26	**1.03**	2.76	3.35	**1.21**	3.99	3.53	0.88
30	2.23	3.16	**1.42**	3.11	3.24	**1.04**	2.71	3.34	**1.23**	3.99	3.59	0.90
40	2.21	3.15	**1.42**	3.10	3.22	**1.04**	2.77	3.35	**1.21**	4.06	3.55	0.87
total	8.95	12.60	**1.41**	12.49	12.96	**1.04**	11.01	13.40	**1.22**	15.98	14.19	0.89

Table 5.7: *Rlg-xwide* warm start performance. The heuristic combination used was simple-deficit-highest-wave-pre-0.

We do not have data describing the sizes of the branches maintained by the pseudoflow

solver. However, we can look at the average number of pushes per merger to approximate the branch sizes involved. A larger number of pushes per merger would tend to indicate larger branches. The average number of pushes per merger for each instance of *rlg-long* and *rlg-xwide* is shown below in Table 5.8.

	a20c20		a20c80		a80c20		a80c80	
seed	long	xwide	long	xwide	long	xwide	long	xwide
10	9.14	1.34	11.91	1.61	8.64	1.58	12.36	1.80
20	8.99	1.36	9.30	1.61	9.23	1.59	13.01	1.80
30	9.31	1.34	9.89	1.60	9.88	1.57	12.96	1.80
40	8.84	1.33	12.84	1.59	8.99	1.57	13.89	1.80

Table 5.8: Average number of pushes per merger for *rlg-long* and *rlg-xwide*.

The extra-wide graphs produced smaller branches than the *rlg-long* graphs. However, the data in Table 5.7 show that this did not significantly improve the performance of the warm start technique. Therefore, we conclude that warm start is not necessarily hampered by large branches.

Graph Size

Another pattern from the scheduling data experiments suggested that warm start performs better on smaller graphs. To test this, we generated *rlg-long* and *rlg-wide* graphs with 2^{13} nodes instead of the graphs with 2^{15} nodes used above. The results are shown in Table 5.9 below.

For the *rlg-long* graphs, the warm start solver is almost always faster than iterative solver on these smaller graphs. However, with the *rlg-wide* graphs, the performance is mixed: in half the cases warm start runs faster, and in the other half it runs slower. Therefore, we cannot conclude that warm start necessarily runs faster on smaller graphs.

Summary

In an effort to validate our observations from the scheduling graphs and to find additional performance patterns, we generated a number of RLG graphs with parameters similar to the scheduling graphs and tested the warm start solver on these generated graphs. However, the experiments with the RLG graphs did not provide conclusive evidence to support the hypotheses that warm start runs faster on either smaller or shorter graphs.

RLG Long												
	a20c20			a20c80			a80c20			a80c80		
seed	warm	iter	ratio	warm	iter	ratio	warm	iter	ratio	warm	iter	ratio
10	0.846	1.488	**1.76**	1.230	2.942	**2.39**	1.118	2.392	**2.14**	1.884	9.334	**4.95**
20	0.780	1.458	**1.87**	1.348	1.498	**1.11**	1.024	3.882	**3.79**	1.790	10.492	**5.86**
30	0.794	1.704	**2.15**	1.398	3.184	**2.28**	0.974	0.874	0.90	1.668	10.214	**6.12**
40	0.796	1.478	**1.86**	1.096	0.964	0.88	1.004	2.798	**2.79**	1.878	4.040	**2.15**
total	3.216	6.128	**1.91**	5.072	8.588	**1.69**	4.120	9.946	**2.41**	7.220	34.080	**4.72**
RLG Wide												
10	0.750	1.012	**1.35**	1.334	1.156	0.87	1.086	1.204	**1.11**	1.976	1.354	0.69
20	0.708	1.032	**1.46**	1.282	0.974	0.76	1.060	1.284	**1.21**	1.866	1.158	0.62
30	0.702	1.036	**1.48**	1.182	0.994	0.84	0.866	0.998	**1.15**	1.694	1.030	0.61
40	0.746	1.088	**1.46**	1.346	1.052	0.78	1.222	1.302	**1.07**	2.002	1.552	0.78
total	2.91	4.17	**1.43**	5.14	4.18	0.81	4.23	4.79	**1.13**	7.54	5.09	0.68

Table 5.9: Performance on smaller RLG graphs (2^{13} nodes). The heuristic combination used was simple-deficit-highest-wave-pre-0.

Furthermore, we could not discern any patterns in the performance based on the number of arcs that changed capacity or the magnitude of the changes.

5.6.3 Mining Data

We conducted the final tests using mining data based on the *mine1* synthetic mining instance. Because we did not concatenate the data file as we did in Chapters 3 and 4, the file contained only 54,000 blocks. Initially, we tested a moderate number of variations of parameters controlling the number of blocks that changed and the magnitude of the changes. We used only a single, default random number seed. The result are shown in Table 5.10.

Instance	warm	iter	ratio
a10c20	10.56	15.62	**1.48**
a20c50	11.16	15.01	**1.34**
a50c20	10.21	15.36	**1.50**
a50c50	9.65	15.04	**1.56**
a20c100	11.93	16.15	**1.35**
total	53.51	77.18	**1.44**

Table 5.10: Performance for warm start and iterative solvers on mining data (54,000 blocks). Random instances were generated from a single, default random number seed. The heuristic combination used was simple-const-highest-lifo-pre-0.

These data suggest that the warm start technique does well in all cases with the

mining data. To verify this, we selected two parameter values, and generated more random instances using specified random number seeds. The results are shown in Table 5.11.

	a20c50			a50c20		
seed	warm	iter	ratio	warm	iter	ratio
10	18.24	15.52	0.85	17.45	15.32	0.88
20	17.69	14.33	0.81	16.11	14.84	0.92
30	18.56	15.61	0.84	16.33	15.91	0.97
40	18.38	15.35	0.84	16.13	15.23	0.94
total	72.87	60.81	0.83	66.02	61.30	0.93

Table 5.11: Performance for warm start and iterative solvers on mining data (54,000 blocks). Random instances were generated using the specified random number seed. The heuristic combination used was simple-const-highest-lifo-pre-0.

The results for this case are uniformly bad: warm start never matches the performance of the iterative solver. Therefore, the overall results for the warm start technique applied to the mining data are mixed, just as it was for the scheduling and RLG instances.

5.7 Summary

In this chapter we presented a new warm start algorithm for use with the pseudoflow algorithm. The technique allows us to solve a series of problem instances where the structure of the graphs is identical, but the capacities of the arcs vary in arbitrary ways that are not controlled by a parameter value. We use the solution from one instance as the basis for the initial, normalized forest to solve the next instance in the series.

The experimental results are inconclusive. In some cases, the warm start algorithm performs better than solving each instance from scratch. However, in other instances it performs worse than solving the instances from scratch. We were unable to discover any consistent explanations or patterns to reliably describe the conditions under which warm start works well or poorly.

Appendix A

Pseudoflow Example

In this appendix, we provide an example of the execution of the first phase of the pseudoflow algorithm on a simple graph.

We will start with the initial graph shown in Figure A.1. The algorithm begins by saturating the arcs out of the source and also the arcs into the sink. This creates nodes with positive excess next to the source and nodes with negative excess next to the sink, as shown in the figure. We call nodes with positive excess *strong* and nodes with non-positive excess *weak*. Nodes that are in the interior of the graph without arcs to either the source or the sink will have zero excess.

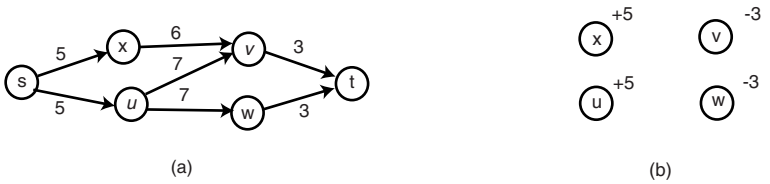


Figure A.1: The initial example graph, and (b) the initial excess and deficit on the nodes.

The algorithm proceeds by sending positive excess from strong nodes to weak nodes along the arcs of the residual graph until no more excess can be sent.

Each iteration begins by finding an arc in the residual graph with positive residual capacity between a strong node and a weak node and then performing a *merger*. Our initial merger will be from u (strong) to v (weak). We call the arc (u, v) a *merger arc*.

The merger creates a *branch* where u is a child of v. The algorithm actually works with branches rather than individual nodes. Single nodes are simply minimal, singleton

branches.

The algorithm requires that branches have excess or deficit only at the root. To restore this property, we attempt to send all five units of the excess from u to v along the merger arc (u, v). In this case, there is sufficient capacity on the arc to send the entire excess of u to v. After pushing the excess, there are two units of residual capacity from u to v and five units back from v to u. The excess of u is now zero, and the excess v is positive two, which is the net of the excess of u and the original deficit of v. This simple merger is shown in Figure A.2.

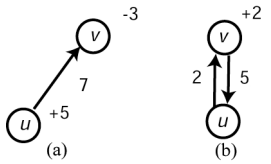


Figure A.2: Simple merger between nodes u and v. (a) node u becomes a child of v, and (b) all of the excess from u is pushed to v.

Because the positive excess of u exceeded the deficit of v, the root of the combined branch has positive excess, so we call it strong, just as we did with the single node. We refer to every node in a strong branch as being strong. For weak branches, if the root has non-positive excess, the branch and all nodes in it are called weak.

Notice that the total number of branches has decreased as a result of the merger—a strong branch and a weak branch have been replaced by one strong branch.

Now, the goal of the algorithm is to find a merger arc from any strong branch to any weak branch. We examine all arcs in the residual graph looking for an arc with positive capacity from a strong node to a weak one. The strong node and the weak node involved in a merger need not be the roots of their respective branches.

To continue our example, we have a residual arc from our original node u to node w. We begin the merger by *rehanging* the strong branch from u, which makes it the root of the branch, and v becomes a child of u. This has reversed the parent-child relationship between u and v, but the residual capacities have not changed, even though we have inverted the branch. Then, we make u a child of w and attempt to push the two units of the excess from v (the former strong root) up to u and across to w. If w were not the root

of the weak branch, we would attempt to push the excess all the way up to the root of the weak branch. In this case, the two units of excess from v is less than the three units of deficit at w, so the merged branch is now weak. See Figure A.3.

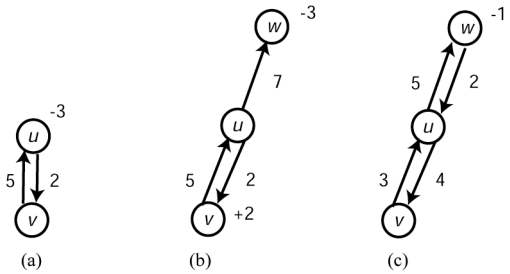


Figure A.3: More complex merger between nodes u and w. (a) the strong branch is rehung. (b) u is made a child of w. (c) The excess from v is pushed to w.

Our example so far has assumed that there is sufficient capacity in the residual network to push the excess from the root of the strong branch all the way to the root of the weak branch. However, this need not be the case. In the next iteration (Figure A.4), we will merge node x, which has five units of excess, to node v. As usual, we make x a child of v and attempt to push all of the excess to the weak root w.

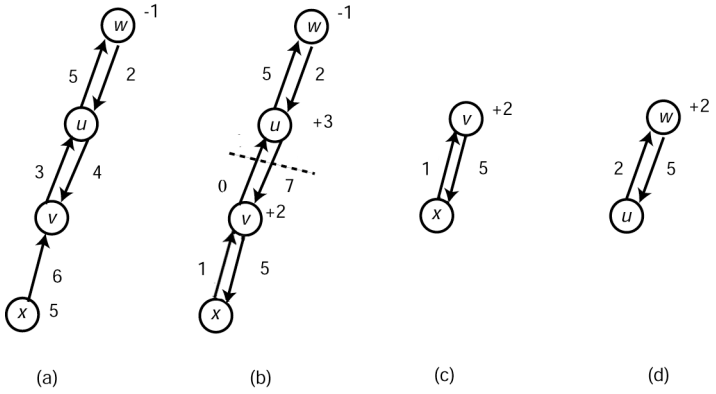


Figure A.4: A merger between x and v with a split operation. (a) x becomes a child of v. (b) five units of excess are pushed from x to v, but only three units can be pushed from v to u. The branch must be split (c) creating a new strong branch with root v with two units of excess. (d) The rest of the excess is pushed from u to w, making it strong.

We can push the excess from x to v, but there is insufficient capacity to push all of the excess from v to u. So, we push what we can, three units, which leaves v with two

units of excess and u with three units. We have saturated the arc (u, v), so we will *split* the branch. This creates a new strong branch with root v which has positive excess equal to the amount of excess that we could not push, two units. We will continue pushing the excess that was pushed to u (three units) towards the weak root. In this case there is sufficient capacity; however that need not be the case. While pushing the flow, the branch can be split multiple times. If the amount of excess exactly matches the residual capacity of the edge, we split the branch.

This merger process continues until we can find no merger arc between a strong node and a weak one. At this point, the first phase of the pseudoflow algorithm is done. However, the pseudoflow is not a feasible flow because the flow balance constraints are violated for the roots of the branches. The pseudoflow can be converted into a feasible flow by a simple process based on flow decomposition that effectively returns excess to the source and deficits to the sink.

Bibliography

[AKMO97] Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. Computational investigations of maximum flow algorithms. *European Journal of Operational Research*, 97(3):509–542, 1997.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

[And97] Charles Anderson. Experiences with extension programming and scripting in Python. In *Proceedings of the 6th International Python Conference*, pages 37–43, San Jose, Ca., October 1997.

[AO89] Ravindra K. Ahuja and James B. Orlin. A fast and simple aogrithm for the maximum flow problem. *Operations Research*, 37:748–759, November 1989.

[AO91] Ravindra K. Ahuja and James B. Orlin. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38:413–430, 1991.

[AOT89] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the maixmum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[AS93] Richard J. Anderson and Joao C. Setubal. Goldberg's algorithm for maximum flow in perspective: a computational study. In *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–17, 1993.

[BB93] T. Badics and E. Boros. Implementing a maximum flow algorithm: Experiments with dynamic trees. In *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–63, 1993.

[BNK98] Amotz Bar-Noy and Guy Kortsarz. Minimum color sum of bipartite graphs. *Journal of Algorithms*, 28(2):339–365, August 1998.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company, 1991.

[BT97] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[CG95a] Craig M. Chase and Vijay K. Garg. Efficient detection of restricted classes of global predicates. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972, pages 303–317, Le Mont-Saint-Michel, France, 1995. Springer-Verlag.

[CG95b] Boris V. Cherkassky and Andrew V. Goldberg. On implementing push-relabel method for the maximum flow problem. In *Proceedings of the 4th International Integer Programming and Optimization Conference*, pages 157–171, May 1995.

[CH89] J. Cheriyan and T. Hagerup. A randomized maximum-flow algorithm. In *30th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 118–123. IEEE Computing Society Press, 1989.

[Che79] Boris V. Cherkassky. A fast algorithm for computing maximum flow in a network. In A. V. Karzanov, editor, *Collected papers, Issue 3: Combinatorial Methods for Flow Problems*, pages 90–96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Trans., Vol. 158.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CM89] J. Cheriyan and S. N. Maheshwari. Analysis of preflow-push algorithms for maximum flow networks. *SIAM Journal on Computing*, 18(6):1057–1086, December 1989.

[Cun85] William H. Cunningham. Optimal attach and reinforcement of a network. *Journal of the ACM*, 32(3):549–561, July 1985.

[Dan63] Geroge B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[DF54] Geroge B. Dantzig and D. R. Fulkerson. Minimizing the number of tankers to meet a fixed schedule. *Naval Research Logistics Quarterly*, 1:217–222, 1954.

[Dim90] The first DIMACS international algorithm implementation challenge: The core experiments, 1990. Available at ftp://dimacs.rutgers.edu/pub/netflow/general-info/core.tex.

[Din70] E. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[DM89] U. Derigs and W. Meier. Implementing goldberg's max-flow-algorithm - a computational investigation. *ZOR - Methods and Models of Operations Research*, 33:383–403, 1989.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[ES76] M. J. Eisner and D. G. Severance. Mathematical techniques for efficient record segmentation in large shared databases. *Journal of the ACM*, 23:619–635, 1976.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

[Eve78] Shimon Even. The max flow algorithm of Dinic and Karzanov: an exposition. In *Information Technology: Proceedings of the 3rd Jerusalem Conference on Information Technology*, pages 233–237, 1978.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FG86] A. Federgruen and H. Groenevelt. Preemptive scheduling of uniform machines by ordinary network flow techniques. *Management Science*, 32:341–349, 1986.

[Fle01] Lisa K. Fleischer. Faster algorithms for the quickest transshipment problem. *SIAM Journal on Optimization*, 12(1):18–35, 2001.

[FSO97] Frederickson and Solis-Oba. Efficient algorithms for robustness in matroid optimization. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, pages 695–678, 1997.

[GC97] Andrew V. Goldberg and Boris V. Cherkassky. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[GG88] D. Goldfarb and Michael D. Grigoriadis. A computational comparison of the Dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13:83–123, 1988.

[GGT89] Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, February 1989.

[GH94] Olivier Goldschmidt and Dorit S. Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. *Mathematics of Operations Research*, 19(1):24–37, February 1994.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides. *Design Patterns*. Addison-Wesley Publishing Company, 1994.

[GKM82] S. L. Graham, P.B. Kessler, and M.K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 120–126, June 1982.

[GM92] D. Gusfield and C. Martel. A fast algorithm for the generalized parametric minimum cut problem and applications. *Algorithmica*, 7:499–519, 1992.

[GM99] Andrew V. Goldberg and Bernard M. E. Moret. Combinatorial algorithm test sets (CATS): The acm/eatcs platform for experimental research. Technical Report 98-130, NEC Research Institute, April 1999.

[Gol84] Andrew V. Goldberg. Fining a maximum density subgraph. Technical Report UCB/CSD/84/171, UC Berkeley, 1984.

[Gol85] Andrew V. Goldberg. *A New Max-Flow Algorithm*. PhD thesis, Massachusetts Institute of Technology, 1985.

[Gol00] Andrew V. Goldberg, 2000. Personal communication.

[GT86] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[GT94] Dan Gusfield and Eva Tardos. A faster parametric minimum-cut algorithm. *Algorithmica*, 11:278–290, 1994.

[HC00] Dorit S. Hochbaum and Anna Chen. Performance analysis and best implementations of old and new algorithms for the open-pit mining problem. *Operations Research*, 48(6):894–914, November 2000.

[Hoc97] Dorit S. Hochbaum. The pseudoflow algorithm: A new algorithm and a new simplex algorithm for the maximum flow problem. Submitted, November 1997.

[Hoc01] Dorit S. Hochbaum. An efficient algorithm for image segmentation, markov random fields and related problems. *Journal of the ACM*, 48(4):686–701, July 2001.

[HP97] Dorit S. Hochbaum and Anu Pathria. Forest harvesting and minimum cuts: A new approach to handling spatial constraints. *Forest Science*, 43:544–554, November 1997.

[JM93] David S. Johnson and Catherine C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1993.

[Joh68] T. B. Johnson. *Optimum Open Pit Mine Production Scheduling.* PhD thesis, University of California, Berkeley, 1968.

[JW86] Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 44–57, 1986.

[Kar74] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[Knu92] Donald Ervin Knuth. *Literate Programming.* Center for the Study of Language and Information, 1992.

[LA98] Ralph Levien and Alex Aiken. Attack-resistant trust metrics for public key certification. In *Proceedings of the 7th USENIX Security Symposium*, pages 229–242, 1998.

[Leh96] Greg Lehey. *Complete FreeBSD.* Walnut Creek CD ROM, June 1996.

[LG65] H. Lerchs and I. F. Grossmann. Optimum design of open-pit mines. *Transactions, C.I.M.*, 68:17–24, 1965.

[LSL99] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA'99*, 1999.

[Lut96] Mark Lutz. *Programming Python.* O'Reilly and Associates, 1996.

[McC99] S. Thomas McCormick. Fast algorithms for parametric scheduling come from extensions to parametric maximum flow. *Operations Research*, 47(5):744–756, September 1999.

[MKM78] 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(6):277–278, October 1978.

[MN00] Kurt Mehlhorn and Stefan Naher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 2000.

[MQW00] Francois Margot, Maurice Queyranne, and Yaoguang Wang. Decompositions, network flows and a precedence constrained single machine scheduling problem. Technical Report 2000-29, Department of Mathematics, University of Kentucky, 2000.

[MSSU03] Rolf H. Möhring, Andreas S. Schultz, Frederik Stork, and Marc Uetz. Solving project scheduling problems by minimum cut computations. *Management Science*, 49(3):330–350, 2003.

[Noe98] Geoffrey J. Noer. Cygwin32: A free win32 porting layer for unix applications. In *Second USENIX Windows NT Symposium, 1998*, 1998.

[NV93] Q. C. Nguyen and V. Venkateswaran. Implementations of the goldberg-tarjan algorithm. In *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–41, 1993.

[Pos88] *IEEE Standard Portable Operating System Interface for Computer Environments*, 1988. IEEE Std 1003.1-1988.

[PSLM00] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library.* Prentice Hall, 2000.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–104, 1994.

[Rog97] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[Rua98] John D. Ruark. *Implementing Reusable Solvers: An Object-Oriented Framework for Operations Research Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1998.

[Sch66] B. L. Schwartz. Possible winners in partially completed tournaments. *SIAM Review*, 8:302–308, 1966.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal Computer Systems Science*, 26:362–391, 1983.

[Sto78] Harold S. Stone. Critical load factors in two-processor distributed systems. *IEEE Transactions on Software Engineering*, 4(3):254–258, May 1978.

[YW94] Hongua Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. In *1994 IEEE/ACM International Conference on Computer-Aided Design*, pages 50–55. IEEE Computer Society Press, 1994.