SOLVING UNWEIGHTED AND WEIGHTED BIPARTITE MATCHING PROBLEMS IN THEORY AND PRACTICE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
J. Robert Kennedy, Jr.
August 1995

Abstract

The push-relabel method has been shown to be efficient for solving maximum flow and minimum cost flow problems in practice, and periodic global updates of dual variables have played an important role in the best implementations. Nevertheless, global updates had not been known to yield any theoretical improvement in running time. In this work, we study techniques for implementing push-relabel algorithms to solve bipartite matching and assignment problems. We show that global updates yield a theoretical improvement in the bipartite matching and assignment contexts, and we develop a suite of efficient cost-scaling push-relabel implementations to solve assignment problems.

For bipartite matching, we show that a push-relabel algorithm using global updates runs in $O\left(\sqrt{n}m\frac{\log(n^2/m)}{\log n}\right)$ time (matching the best bound known) and performs worse by a factor of $\sqrt{n}$ without the updates. We present a similar result for the assignment problem, for which an algorithm that assumes integer costs in the range $[-C,\ldots,C]$ runs in time $O(\sqrt{n}m\log(nC))$ (matching the best cost-scaling bound known).

We develop cost-scaling push-relabel implementations that take advantage of the assignment problem's special structure, and compare our codes against the best codes from the literature. The results show that the push-relabel method is very promising for practical use.

Preface

As a visible product of my time spent in graduate school, this thesis is an opportunity for me to thank a few of the great many people who have shaped my experience over the last several years. Many have contributed directly to my academic progress, and arguably some have detracted from it. But academic progress is not everything, and each of the people I will name (as well as each who merits a place in these pages but whom I inadvertently omit) has influenced me for the better, and has made me a fuller, happier person.

First, I thank my principal advisor Andrew Goldberg, to whom this work owes a great debt of natures both technical and inspirational. Without Andrew's work on push-relabel algorithms, the questions this thesis answers could not have been asked. During the development of the results herein, too, Andrew has played a major technical role as my collaborator in most of the work. Also of great importance have been the contributions he has made by suggesting directions for research, his understanding of the research community, and his constructive suggestions at times when I wasn't sure how to proceed. Many of my student colleagues have given me the impression that with or without an advisor, they would finish the Ph.D. program having done excellent work. I am not such a student, though, and the completion of this work says a great deal about Andrew's capacity to advise a student who really needs an advisor.

Other faculty members and students have made invaluable contributions to my understanding of computer science during my years at Stanford. Worthy of special mention are my other two readers, Serge Plotkin and Rajeev Motwani; Donald Knuth; Vaughan Pratt; Leo Guibas; Steven Phillips; Tomek Radzik, and David Karger.

Though they probably don't realize what an impression they made, Sanjeev Khanna and Liz Wolf reminded me at pivotal times that I could complete the Ph.D. program, and I owe them special gratitude for this as well as for their friendship. My Hoang and Ashish Gupta never hesitated to welcome me into conversation when I needed to talk.

For their vital parts in my life outside of computer science, I thank Bill Bell, Jim Nadel, Cathy Haas, Lon Kuntze, Steven Phillips, Nella Stoltz-Phillips, Roland Conybeare, Rita Battaglin, Arul Menezes, Patrick Lincoln, Alan Hu, Eric Torng, Susan Kropf, Jim Franklin, Russ Haines, Tracy Schmidt, Karen Pieper, Laure Humbert, Scott Burson, the members (past and present) of the `cycles` and `tuesday` mailing lists, and all those others whose names I have forgotten to include.

For their constant support and encouragement beginning the day I was born and continuing steadfastly through the present, I owe an inexpressible debt to my parents Liz and Jim Kennedy. Their contribution to making me a whole person has consistently been greater than anyone could expect and is certainly deeper than anyone knows; they are models for me as people and, should I father a child someday, as parents.

A brother's influence is harder to express clearly, but anyone who knows me knows that my brother William has molded me and affected me profoundly. Part of him goes with me always, and contributes to everything I do. I am particularly grateful to him for introducing me to Stanford — a place I have loved all my time here — and to many of Stanford's people.

As my partner in life through the years during which I did the work represented here, Lois Steinberg has supported me in ways I'm sure I don't even realize, as well as the numerous and giant ways I'm aware of. She has encouraged me, advised me, buoyed my spirits, taught me valuable lessons, listened to me intently, and given of herself constantly. Her generosity with her smile and her love keeps me going through challenging times, and makes possible a great many worthwhile things I could not otherwise do.

Finally on a more technical note, this work owes specific debts to a few colleagues whom I acknowledge here. I would like to thank David Castañon for supplying and assisting with the SFR10 code, Anil Kamath and K. G. Ramakrishnan for their assistance in interpreting results reported in [48], Jianxiu Hao for supplying and assisting with the SJV and DJV implementations, Serge Plotkin for his help producing the digital pictures, and Scott Burson for sharing his home and providing computing facilities during the development of the codes.

My apologies and my request for forgiveness go to those deserving people whom I've left out. If, upon reading these words you wonder whether perhaps you belong among them, you probably do.

Contents

List of Figures

Chapter 1

Introduction

Problems of network flow and their close relatives fall in the surprisingly small class of graph problems for which efficient algorithms are known, but that seem to require more than linear time. Their position on this middle ground of tractability and their aesthetic appeal account for a great deal of the theoretical interest in them in the past decades. Further, network flow problems are of practical interest because they abstract important characteristics of many problems that arise in other, less theoretical, domains.

Algorithms to solve network flow problems fall roughly into three main categories. The first two classes of algorithms take advantage of the fact that network flow problems are special cases of linear programming. The simplex method and closely allied techniques for general linear programming and linear programming on networks (see [12] or [10] for an introduction) form the first group. The second group is another class of algorithms capable of solving general linear programs, namely interior point methods (see [49] for an overview). In the final group are algorithms that exploit specific aspects of the combinatorial structure of network flow problems, that run in polynomial time, and that are not designed as specializations of techniques for general linear programming.

A great deal of effort has been devoted to the task of making simplex methods run fast, both in theory and in practice, on various special types of combinatorial problems, including flow problems ([3, 11, 33, 26, 34, 44, 51] is a small sampling of the literature in this area).

Although simplex and interior-point methods remain attractive to practitioners who solve general linear programming problems, they are theoretically inferior in the network flow context to combinatorial algorithms. Relatively recent research has shown that for many broad classes of network flow problem instances the general techniques, even specialized to particular network flow problem formulations, are inferior to polynomial-time combinatorial algorithms in practice as well [2, 6, 8, 14, 25, 39, 45]. Our work widens the gap by which combinatorial algorithms win out over general linear programming algorithms in practice, and provides theoretical analysis of a heuristic that has been shown to improve the performance of polynomial-time combinatorial codes.

In this work we focus on a family of combinatorial algorithms for network flow and their application to a specific class of network optimization problems. Specifically, we study theoretical and practical properties of particular combinatorial algorithms for assignment and closely related problems, and we evaluate several heuristics for implementing these algorithms. We develop a suite of implementations and compare their performance against other codes from the literature. Throughout, we stress the interplay between theory and practice: we prove useful theoretical properties of practical heuristics and we develop codes that perform well in practice and that obey good theoretical time bounds.

The bipartite matching problem is a classical object of study in graph algorithms, and has been investigated for several decades (an early reference is [35]). Informally, the problem models the situation in which there are two types of objects, say college students and available slots in dormitory rooms, and each student provides a list of those slots that s/he would find acceptable. Solving the bipartite matching problem consists of determining as large a set as possible of pairings of students to slots that meet the students' acceptability constraints. The assignment problem (or weighted bipartite matching) has also been studied for several decades, and incorporates the consideration of costs on the edges of the problem graph. It has applications in resource allocation, image processing, handwriting recognition, and many other areas; moreover, it arises as a subproblem in algorithms to solve much more difficult optimization problems such as the Traveling Salesman Problem [17]. Like many combinatorial optimization problems, bipartite matching and assignment can be viewed as special types of network flow problems.

Of major importance in recent algorithms for network flow is the push-relabel framework of Goldberg and Tarjan [31, 32]. A push-relabel algorithm for a network flow problem works by maintaining tentative primal and dual solutions (to use the terminology of linear programming). The tentative solutions are updated locally via the elementary operations *push* and *relabel*, and after polynomial time the algorithm terminates with an optimum primal solution.

The push-relabel method is the best currently known way for solving the maximum flow problem in practice [2, 14, 45]. This method extends to the minimum cost flow problem using cost-scaling [24, 32], and an implementation of this technique has proven very competitive on a wide class of problems [25] (see also [29]). In both contexts, the heuristic of periodic global updates of node distances or prices has been critical to obtaining the best running times in practice.

In this thesis, we investigate a variety of heuristics that can be applied to push-relabel algorithms to solve bipartite matching and assignment problems in practice. Some of these techniques have been used in the literature to solve minimum cost flow problems, some are similar to known methods for assignment problems, and some are new.

One of the main contributions of this work is a theoretical analysis of the global update heuristic that has proven so beneficial in implementations. Our analysis shows that the heuristic can improve the theoretical efficiency of push-relabel algorithms as well as their practical performance, and thus represents a step toward formal justification of the heuristic. The literature contains other instances in which heuristics developed for practical implementations have proven theoretically valuable. For example, Hao and Orlin [36] showed that a heuristic for push-relabel maximum flow algorithms gave a substantial improvement in an algorithm to compute the overall minimum cut in a capacitated graph.

We continue with a brief overview of past research on the problems we study, as well as an outline of our contributions. Most of the research in this thesis is joint work with Andrew V. Goldberg; portions of the work have been published in [27] and [28].

1.1 Bipartite Matching

1.1.1 History and Related Work

Several algorithms for bipartite matching run in $O(\sqrt{n}m)$ time.[1] Hopcroft and Karp [37] first proposed an algorithm that achieves this bound.

Karzanov [41, 40] and Even and Tarjan [18] proved that the blocking flow algorithm of Dinitz [16] runs in $O(\sqrt{n}m)$ time when applied to the bipartite matching problem.

Two-phase algorithms based on a combination of the push-relabel method [31] and the augmenting path method [20] were proposed in [30, 46].

Feder and Motwani [19] gave a graph compression technique that combines as a preprocessing step with the algorithm of Dinitz or with that of Hopcroft and Karp to yield an $O\left(\sqrt{n}m\frac{\log(n^2/m)}{\log n}\right)$ algorithm for bipartite matching. This is the best time bound known for the problem.

1.1.2 Our Contributions

Our treatment begins by showing that in the appropriate algorithmic circumstances, a heuristic that periodically updates the dual solution in a global way provides a substantial gain in theoretical performance. We analyze the theoretical properties of such global updates in detail, and demonstrate that they lead to a provable gain in asymptotic performance. Our bipartite matching algorithm using global updates runs in time $O\left(\sqrt{n}m\frac{\log(n^2/m)}{\log n}\right)$, equaling the best sequential time bound known. This theoretical result uses a new selection strategy for push-relabel algorithms; we call our scheme the *minimum distance* strategy. We prove that without global updates, our bipartite matching algorithm performs significantly worse.

[1]Throughout this thesis, n and m denote the number of nodes and edges, respectively, in the input problem.

Results similar to those we obtain for bipartite matching can be derived for maximum flows in networks with unit capacities.

1.2 The Assignment Problem

1.2.1 History and Related Work

The first algorithm developed specifically for solving the assignment problem was the classical "Hungarian Method" proposed by Kuhn [43]. Kuhn's algorithm has played an important part in the study of linear optimization problems, having been generalized to the Primal-Dual method for Linear Programming [13]. The Hungarian Method has the best currently known strongly polynomial time bound[2] of $O(n(m + n \log n))$. Under the assumption that the input costs are integers in the range $[-C, \dots, C]$, Gabow and Tarjan [23] use cost-scaling and blocking flow techniques to obtain an $O(\sqrt{n}m \log(nC))$ time algorithm. An algorithm using an idea similar to global updates with the same running time appeared in [22]. Two-phase algorithms with the same running time appeared in [30, 46]. The first phase of these algorithms is based on the push-relabel method and the second phase is based on the successive augmentation approach.

1.2.2 Our Contributions

We present an algorithm for the assignment problem that makes use of global updates in a weighted context. Our algorithm runs in $O(\sqrt{n}m \log(nC))$ time, and like the other algorithms with this time bound, it is based on cost-scaling, assumes the input costs are integers, and is not strongly polynomial. This bound is the best cost-scaling bound known, and under the similarity assumption (*i.e.*, the assumption that $C = O(n^k)$ for some constant k) is the best bound known for the problem. Ours is the first algorithm to achieve this time bound with "single-phase" push-relabel scaling

[2]All the algorithms we discuss run in time polynomial in n, m, and $\log C$, where C is a bound on the largest magnitude of an edge weight; for an algorithm on weighted graphs to qualify as *strongly polynomial*, its running time must be polynomially bounded in only n and m, independent of the arc costs.

iterations; the single-phase structure allows us to avoid building parameters from the analysis into the algorithm itself. Global updates are crucial in obtaining our performance bounds; we prove that the same algorithms without the heuristic perform asymptotically worse. These results represent a step toward a theoretical understanding of the global update heuristic's contribution to push-relabel implementations.

Our selection scheme for our theoretical work on the assignment problem is the *minimum price change* strategy. Like the minimum distance strategy for bipartite matching, this scheme is new to the development of push-relabel algorithms.

Similar results can be obtained for minimum cost flows in networks with unit capacities.

After the theoretical development, we briefly investigate the effects of global updates on practical performance when we apply a push-relabel implementation for minimum cost flow to a battery of assignment problems.

Following our discussion of global updates, we outline several additional techniques that can be applied to improve the performance of implementations that solve the assignment problem, and we briefly discuss the characteristics of each. Although the heuristics other than global updates are not known to change the algorithms' asymptotic running times, they allow various improvements in practical efficiency which we describe.

In the final part of the thesis, we develop a suite of three implementations specialized to the assignment problem. The codes use several of the ideas we have discussed and perform well on a broad class of assignment problems. We investigate the performance of these codes in detail by supplying them with problem instances from a variety of generators and timing them in a standardized computing environment; we compare them on the basis of running time with with the best codes from the literature, and observe that our best implementation is robust and usually runs faster than its competitors.

Chapter 2

Background on Network Flow Problems

This chapter describes the four combinatorial optimization problems we will encounter, and develops some basic notation for dealing with them. We will also briefly survey algorithmic background information before describing the generic push-relabel framework as it applies to these problems.

2.1 Problems and Notation

2.1.1 Bipartite Matching and Maximum Flow

Let $\overline{G} = (\overline{V} = X \cup Y, \overline{E})$ be an undirected bipartite graph, let $n = |\overline{V}|$, and let $m = |\overline{E}|$. A *matching* in $\overline{G}$ is a subset of edges $M \subseteq \overline{E}$ that have no node in common. The *cardinality* of the matching is $|M|$. The *bipartite matching problem* is to find a matching of maximum cardinality.

Several algorithms have been proposed for solving the bipartite matching problem that work directly with the matching formulation. In particular, Hopcroft and Karp's algorithm [37] has historically been expressed from a matching point of view, and was the first algorithm to achieve a running time of $O(\sqrt{n}m)$.

Unlike Hopcroft and Karp's classic result, a great many algorithms for bipartite matching are best understood through a reduction to the maximum flow problem. First we introduce the maximum flow problem and its notation, and then we proceed with a description of the standard reduction from bipartite matching to maximum flow.

The conventions we assume for the maximum flow problem are as follows: Let $G = (\{s,t\} \cup V, E)$ be a digraph with an integer-valued *capacity* $u(a)$ associated with each arc[1] $a \in E$. We assume that $a \in E \Longrightarrow a^R \in E$ (where a^R denotes the reverse of arc a).

A *pseudoflow* is a function $f : E \rightarrow \mathbf{R}$ satisfying the following for each $a \in E$:

- $f(a) = -f(a^R)$ (*flow antisymmetry* constraints);
- $f(a) \leq u(a)$ (*capacity* constraints).

The antisymmetry constraints are for notational convenience only, and we will often take advantage of this fact by mentioning only those arcs with nonnegative flow; in every case, the antisymmetry constraints are satisfied simply by setting each reverse arc's flow to the appropriate value. For a pseudoflow f and a node v, the *excess flow into* v, denoted $e_f(v)$, is defined by $e_f(v) = \sum_{(u,v)\in E} f(u,v)$. A *preflow* is a pseudoflow with the property that the excess of every node except s is nonnegative. A node $v \neq t$ with $e_f(v) > 0$ is called *active*.

A *flow* is a pseudoflow f such that, for each node $v \in V$, $e_f(v) = 0$. Observe that a preflow f is a flow if and only if there are no active nodes. The *maximum flow problem* is to find a flow maximizing $e_f(t)$.

We reduce the bipartite matching problem to the maximum flow problem in a standard way. For brevity, we mention only the "forward" arcs in the flow network; to each such arc we give unit capacity. The "reverse" arcs have capacity zero. Given an instance $\overline{G} = (\overline{V} = X \cup Y, \overline{E})$ of the bipartite matching problem, we construct an instance $(G = (\{s,t\} \cup V, E), u)$ of the maximum flow problem by

- setting $V = \overline{V}$;

[1]Sometimes we refer to an arc a by its endpoints, *e.g.*, (v, w). This is ambiguous if there are multiple arcs from v to w. An alternative is to refer to v as the tail of a and to w as the head of a, which is precise but inconvenient.

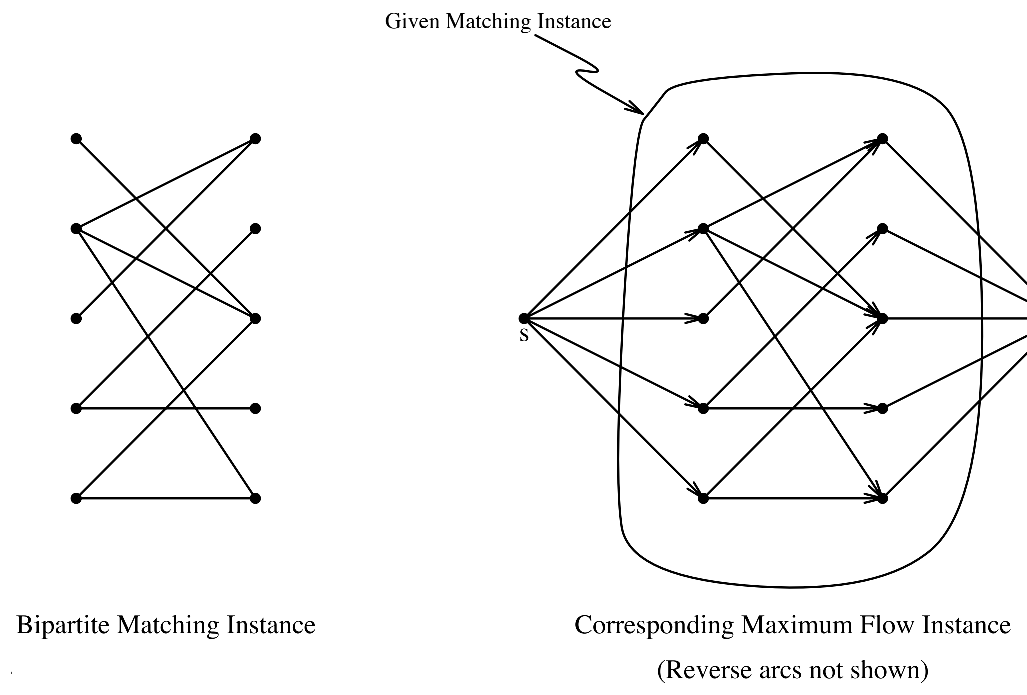

Given Matching Instance
s
t
Bipartite Matching Instance
Corresponding Maximum Flow Instance
(Reverse arcs not shown)

Figure 2.1: Reduction from Bipartite Matching to Maximum Flow

- for each node $v \in X$ placing arc (s, v) in E;
- for each node $v \in Y$ placing arc (v, t) in E;
- for each edge $\{v, w\} \in \overline{E}$ with $v \in X$ and $w \in Y$ placing arc (v, w) in E.

A graph obtained by this reduction is called a *matching network*. Note that if G is a matching network, then for any integral pseudoflow f and for any arc $a \in E$, $u(a), f(a) \in \{0, 1\}$. Indeed, any integral flow in G can be interpreted conveniently as a matching in $\overline{G}$: the matching is exactly the edges corresponding to those arcs $a \in X \times Y$ with $f(a) = 1$. It is a well-known fact [20] that a maximum flow in G corresponds to a maximum matching in $\overline{G}$.

2.1.2 Assignment and Minimum Cost Circulation Problems

Given a weight function $\overline{c} : \overline{E} \to \mathbf{R}$ and a set of edges M, we define the weight of M to be the sum of weights of edges in M. The *assignment problem* is to find a maximum cardinality matching of minimum weight in a bipartite graph. We assume that the costs are integers in the range $[0,\ldots,C]$ where $C \geq 1$. (Note that we can always make the costs nonnegative by adding an appropriate number to all arc costs, and without affecting the asymptotic time bounds we claim.)

For the minimum cost circulation problem, we adopt the following framework. We are given a graph $G = (V, E)$, with an integer-valued capacity function as in the case of maximum flow. In addition to the capacity function, we are given an integer-valued *cost* $c(a)$ for each arc $a \in E$.

We assume $c(a) = -c(a^R)$ for every arc a. A *circulation* is a pseudoflow f with the property that $e_f(v) = 0$ for every node $v \in V$. (The absence of a distinguished source and sink accounts for the difference in nomenclature between a circulation and a flow.) We will say that a node v with $e_f(v) < 0$ has a *deficit*.

The cost of a pseudoflow f is given by $c(f) = \sum_{f(a)>0} c(a)f(a)$. The *minimum cost circulation problem* is to find a circulation of minimum cost.

We reduce the assignment problem to the minimum cost circulation problem as follows. As in the unweighted case, we mention only "forward" arcs, each of which we give unit capacity. The "reverse" arcs have zero capacity and obey cost antisymmetry. Given an instance $(\overline{G} = (\overline{V} = X \cup Y, \overline{E}), \overline{c})$ of the assignment problem, we construct an instance $(G = (\{s,t\} \cup V, E), u, c)$ of the minimum cost circulation problem by

- creating special nodes s and t, and setting $V = \overline{V} \cup \{s,t\}$;
- for each node $v \in X$ placing arc (s,v) in E and defining $c(s,v) = -nC$;
- for each node $v \in Y$ placing arc (v,t) in E and defining $c(v,t) = 0$;
- for each edge $\{v,w\} \in \overline{E}$ with $v \in X$ placing arc (v,w) in E and defining $c(v,w) = \overline{c}(v,w)$;
- placing $n/2$ arcs (t,s) in E and defining $c(t,s) = 0$.

If G is obtained by this reduction, we can interpret an integral circulation in G as a matching in $\overline{G}$ just as we did in the bipartite matching case. Further, it is easy to verify that a minimum cost circulation in G corresponds to a maximum matching of minimum weight in $\overline{G}$. No confusion will arise from the fact that a graph obtained through this reduction, although slightly different in structure from the graphs of

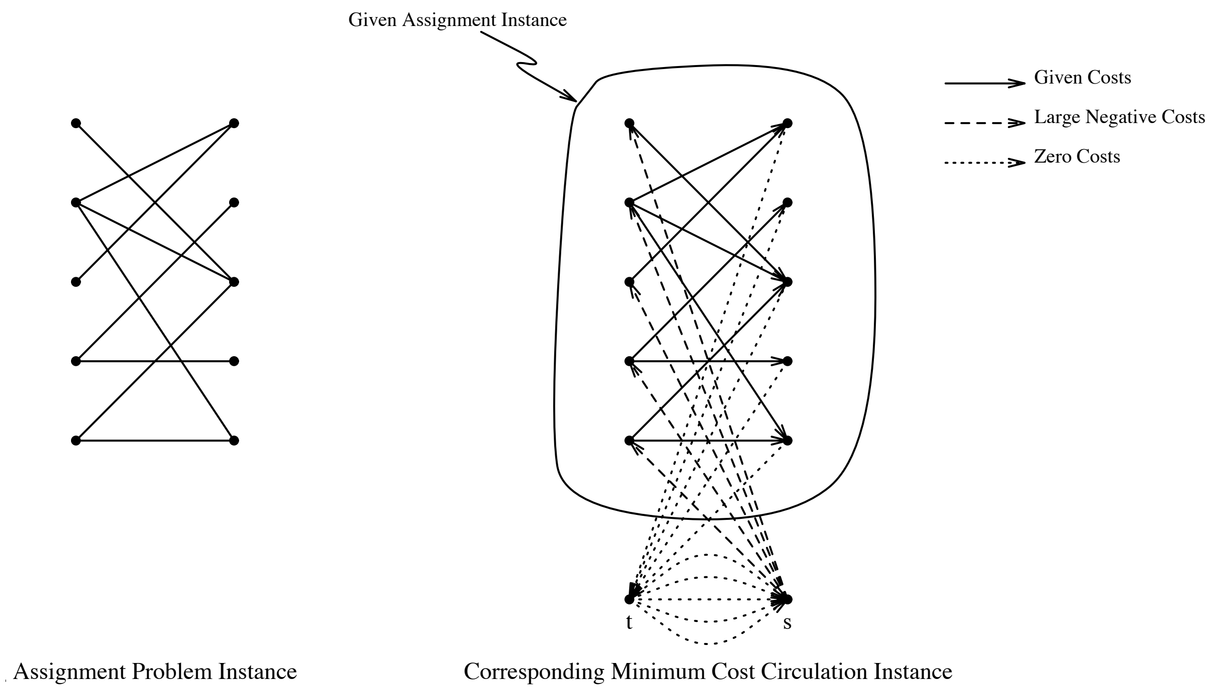

Given Assignment Instance
Given Costs
Large Negative Costs
Zero Costs
t
s
Assignment Problem Instance
Corresponding Minimum Cost Circulation Instance

Figure 2.2: Reduction from Assignment to Minimum Cost Circulation

Section 2.1.1, is also called a matching network.

2.2 The Generic Push-Relabel Framework

Push-relabel algorithms solve both unweighted [24, 31] and weighted [24, 32] network flow problems. In this chapter, we acquaint the reader with unit-capacity versions of these algorithms, since all of our results pertain to networks with unit capacities. See [31, 32] for details of the algorithms with general capacities.

For a given pseudoflow f, the *residual capacity* of an arc $a \in E$ is $u_f(a) = u(a) - f(a)$. The set E_f of *residual arcs* contains the arcs $a \in E$ with $f(a) < u(a)$. The *residual graph* $G_f = (V, E_f)$ is the graph induced by the residual arcs. Those arcs not in the residual graph are said to be *saturated*.

```
RELABEL($v$).
    replace $d(v)$ by $\min_{(v,w)\in E_f}\{d(w)+1\}$
end.
```

```
PUSH($v, w$).
    send a unit of flow from $v$ to $w$.
end.
```

Figure 2.3: The *push* and *relabel* operations

2.2.1 Maximum Flow in Unit-Capacity Networks

A *distance labeling* is a function $d : V \to \mathbf{Z}^+$. We say a distance labeling d is *valid* with respect to a pseudoflow f if $d(t) = 0$, $d(s) = n$, and for every arc $(v, w) \in E_f$, $d(v) \leq d(w)+1$. Those residual arcs (v, w) with the property that $d(v) = d(w)+1$ are called *admissible* arcs, and the *admissible graph* $G_A = (V, E_A)$ is the graph induced by the admissible arcs. It is straightforward to see that G_A is acyclic for any valid distance labeling.

We begin with a high-level description of the generic push-relabel algorithm for maximum flow specialized to the case of networks in which all arc capacities are zero or one. The algorithm starts with the zero flow, then sets $f(a) = 1$ for every arc a of the form (s, v). For an initial distance labeling, the algorithm sets $d(s) = n$ and $d(t) = 0$, and for every $v \in V$, sets $d(v) = 0$. Then the algorithm applies *push* and *relabel* operations in any order until the current pseudoflow is a flow. The *push* and *relabel* operations, described below, preserve the properties that the current pseudoflow f is a preflow and that the current distance labeling d is valid with respect to f.

The *push* operation applies to an admissible arc (v, w) whose tail node v is active. It consists of "pushing" a unit of flow along the arc, *i.e.*, increasing $f(v, w)$ by one, increasing $e_f(w)$ by one, and decreasing $e_f(v)$ by one. The *relabel* operation applies to an active node v that is not the tail of any admissible arc. It consists of changing v's distance label so that v is the tail of at least one admissible arc, *i.e.*, setting $d(v)$ to the largest value that preserves the validity of the distance labeling. See Figure 2.3.

Our analysis of the push-relabel method is based on the following facts. See [31] for details; note that the operations maintain integrality of the current preflow, so every *push* operation saturates an arc.

- For all nodes v, we have $0 \leq d(v) \leq 2n$.
- Distance labels do not decrease during the computation.
- RELABEL(v) increases $d(v)$.
- The number of *relabel* operations during the computation is $O(n)$ per node.
- The work involved in *relabel* operations is $O(nm)$.
- If a node v is relabeled t times during a computation segment, then the number of pushes from v is at most $(t+1) \times \mathit{degree}(v)$.
- The number of *push* operations during the computation is $O(nm)$.

The above facts imply that any push-relabel algorithm runs in $O(nm)$ time given that the work involved in selecting the next operation to apply does not exceed the work involved in applying these operations. This can be easily achieved using the following simple data structure (see [31] for details). We maintain a *current arc* for every node. Initially the first arc in the node's arc list is current. When pushing flow excess out of a node v, we push it on v's current arc if the arc is admissible, or advance the current arc to the next arc on the arc list. When there are no more arcs on the list, we relabel v and set v's current arc to the first arc on v's arc list.

2.2.2 The Push-Relabel Method for the Assignment Problem

A *price function* is a function $p : V \to \mathbf{R}$. For a given price function p, the *reduced cost* of an arc (v, w) is $c_p(v, w) = p(v) + c(v, w) - p(w)$ and the *partial reduced cost* is $c'_p(v, w) = c(v, w) - p(w)$.

Let $U = X \cup \{t\}$. Note that all arcs in E have one endpoint in U and one endpoint in its complement. Define E_U to be the set of arcs whose tail node is in U.

It is common to view the set of node prices as variables in the linear programming dual of the minimum cost flow problem (for more information on linear programming and duality, see [10] or [47]). Linear programming theory provides a set of conditions called *complementary slackness* that are necessary and sufficient for a primal (flow) and dual solution (price function) to be optimum for their respective problems. In the case of minimum cost circulation, the complementary slackness conditions say that if an arc $a \in E_f$, then $c_p(a) \geq 0$, *i.e.*, that there are no residual arcs with negative reduced cost.

A cost-scaling push-relabel algorithm uses the notion of *approximate optimality.* The algorithm generates a sequence of (flow, price function) pairs, corresponding to smaller and smaller values of an *error parameter*, usually denoted by ϵ. The error parameter for a pair is a bound on the negativity of the reduced cost of any residual arc in the network; the flow and price function are guaranteed to violate the complementary slackness conditions by only a limited amount.

In push-relabel algorithms that solve the minimum cost flow problem in its full generality, the notion of approximate optimality typically takes the following form: We say that a pseudoflow f is *ϵ-optimal with respect to a price function p* if every residual arc $a \in E_f$ obeys $c_p(a) \geq -\epsilon$.

In algorithms specialized to the assignment problem we will find it useful to define approximate optimality slightly differently. We will sometimes refer to the following as an *asymmetric* definition of approximate optimality. For a constant $\epsilon \geq 0$, a pseudoflow f is said to be *ϵ-optimal with respect to a price function p* if, for every residual arc $a \in E_f$, we have

$$\begin{cases} a \in E_U \Longrightarrow c_p(a) \geq 0, \\ a \notin E_U \Longrightarrow c_p(a) \geq -2\epsilon. \end{cases}$$

A pseudoflow f is *ϵ-optimal* if f is ϵ-optimal with respect to some price function p. Under either definition of ϵ-optimality, if the arc costs are integers and $\epsilon < 1/n$, any ϵ-optimal circulation is optimal [4, 32].[2]

We have already seen in the case of maximum flow that as a push-relabel algorithm works to improve the current primal and dual solutions, it uses a notion of which arcs are eligible to carry an increased amount of flow. As with approximate optimality,

[2]A minor technicality arises because we have assumed n to be the number of nodes in the given assignment problem, rather than the number in the minimum cost flow instance resulting from our reduction. Since the reduction adds two nodes to the graph, we may either substitute $n + 2$ for n in the foregoing statement or we may make the technical assumptions that $n \geq 2$ and $nC \geq 4$. For the remainder of the thesis, we will neglect these inconsequential details.

various definitions of arc eligibility are possible when the arcs have costs. Push-relabel algorithms for general minimum cost flow typically use the following: An arc $a \in E_f$ is said to be *admissible* if $c_p(a) < 0$.

In algorithms for the assignment problem we will use the following more specialized definition, and call it an *asymmetric* definition of admissibility: For a given f and p, an arc $a \in E_f$ is *admissible* if

$$\begin{cases} a \in E_U \text{ and } c_p(a) < \epsilon & \text{or} \\ a \notin E_U \text{ and } c_p(a) < -\epsilon. \end{cases}$$

The *admissible graph* $G_A = (V, E_A)$ is the graph induced by the admissible arcs.

These asymmetric definitions of ϵ-optimality and admissibility are natural in the context of the assignment problem. They have the benefit that any pseudoflow violates the complementary slackness conditions on $O(n)$ arcs (corresponding essentially to the matched arcs). For the symmetric definition, complementary slackness can be violated on $\Omega(m)$ arcs. This property turns out to be important for technical reasons underlying the proof of Lemma 3.5.5. Throughout this thesis, our algorithms will use the asymmetric definitions of ϵ-optimality and admissibility except where we explicitly state otherwise.

Now we give a high-level description of the successive approximation algorithm (see Figure 2.4). The algorithm starts with $\epsilon = C$, $f(a) = 0$ for all $a \in E$, and $p(v) = 0$ for all $v \in V$. At the beginning of every iteration, the algorithm divides ϵ by a constant factor α and saturates all arcs a with $c_p(a) < 0$. The iteration modifies f and p so that f is a circulation that is (ϵ/α)-optimal with respect to p. When $\epsilon < 1/n$, f is optimal and the algorithm terminates. The number of iterations of the algorithm is $1 + \lfloor \log_\alpha(nC) \rfloor$.

Reducing ϵ is the task of the subroutine *refine*. The input to *refine* is ϵ, f, and p such that (except in the first iteration) circulation f is ϵ-optimal with respect to p. The output from *refine* is $\epsilon' = \epsilon/\alpha$, a circulation f, and a price function p such that f is ϵ'-optimal with respect to p. At the first iteration, the zero flow is not C-optimal with respect to the zero price function, but because every simple path in the residual graph has length of at least $-nC$, standard results about *refine* remain true.

```
procedure MIN-COST($V, E, u, c$);
    [initialization]
    $\epsilon \leftarrow C$ ; $\forall v, \quad p(v) \leftarrow 0; \quad \forall a, \quad f(a) \leftarrow 0$;
    [loop]
    while $\epsilon \geq 1/n$ do
        $(\epsilon, f, p) \leftarrow refine(\epsilon, f, p)$;
    return($f$);
end.
```

Figure 2.4: The cost-scaling algorithm.

```
procedure REFINE($\epsilon, f, p$);
    [initialization]
    $\epsilon \leftarrow \epsilon/\alpha$;
    $\forall a \in E$ with $c_p(a) < 0, \quad f(a) \leftarrow u(a)$;
    [loop]
    while $f$ is not a circulation
        apply a push or a relabel operation;
    return($\epsilon, f, p$);
end.
```

Figure 2.5: The generic *refine* subroutine.

The generic *refine* subroutine (described in Figure 2.5) begins by decreasing the value of ϵ, and setting f to saturate all residual arcs with negative reduced cost.

This converts f into an ϵ-optimal pseudoflow (indeed, into a 0-optimal pseudoflow). Then the subroutine converts f into an ϵ-optimal circulation by applying a sequence of *push* and *relabel* operations, each of which preserves ϵ-optimality. The generic algorithm does not specify the order in which these operations are applied. Next, we describe the *push* and *relabel* operations for the unit-capacity case.

As in the maximum flow case, a *push* operation applies to an admissible arc (v, w) whose tail node v is active, and consists of pushing one unit of flow from v to w. A *relabel* operation applies to an active node v that is not the tail of any admissible arc. The operation sets $p(v)$ to the smallest value allowed by the ϵ-optimality constraints, namely $\max_{(v,w) \in E_f}\{p(w) - c(v, w)\}$ if $v \in U$, or $\max_{(v,w) \in E_f}\{p(w) - c(v, w) - \epsilon\}$

```
RELABEL($v$).
    if $v \in U$
        then replace $p(v)$ by $\max_{(v,w)\in E_f}\{p(w) - c(v,w)\}$
        else replace $p(v)$ by $\max_{(v,w)\in E_f}\{p(w) - c(v,w) - 2\epsilon\}$
end.

PUSH($v, w$).
    send a unit of flow from $v$ to $w$.
end.
```

Figure 2.6: The *push* and *relabel* operations

otherwise.

The analysis of cost-scaling push-relabel algorithms is based on the following facts [30, 32]. During a scaling iteration

- no node price increases;
- every relabeling decreases a node price by at least ϵ;
- for any $v \in V$, $p(v)$ decreases by $O(n\epsilon)$.

Chapter 3

Global Updates: Theoretical Development

As we have seen in Chapter 2, push-relabel algorithms work by maintaining for each node an estimate of the distance (or cost) that must be traversed to arrive at a "destination" from that node. Global updates are a technique for periodically making that estimate exact. The motivation behind global updates as an implementation heuristic is that by ensuring that the admissible graph contains a path from every excess to a sink, they tend to reduce the number of *push* and *relabel* operations performed by the algorithm. Intuitively it is plausible that the more directly the admissible graph "guides" excesses to their destination(s), the better the performance we ought to expect. One way of specifying this directness is to say that after a global update, every step taken by a unit of excess in the admissible graph must make some "irrevocable progress" toward a deficit. To ensure such a condition, we will establish that global updates do not introduce admissible cycles, nor do they leave "dead-ends" in the parts of the admissible graph that might be encountered by an excess. Lastly, to be useful global updates must run efficiently and will have to preserve the basic analysis of the push-relabel method as well. These properties make sense from a practical perspective, and would be desirable in an implementation regardless of their theoretical consequences.

In this chapter, we will show that the above properties of global updates lead to positive theoretical results as well. We will formalize each of the properties and show that they lead to improved bounds on the running times of some push-relabel algorithms. In the bipartite matching case, global updates and the structure of the admissible graph are simple enough that the required properties will obviously hold; for the assignment problem we formalize them in Theorems 3.6.7, 3.6.8, and 3.6.9.

This chapter is organized as follows. In Section 3.1, we present an $O(\sqrt{n}m)$ time bound for the bipartite matching algorithm with global updates, and in Section 3.2 we show how to apply Feder and Motwani's techniques to improve the algorithm's performance to $O\left(\sqrt{n}m\frac{\log(n^2/m)}{\log n}\right)$. Section 3.3 shows that without global updates, the bipartite matching algorithm performs poorly. In Section 3.4, we briefly study the practical effects of global updates in solving bipartite matching problems. Sections 3.5 and 3.7 generalize the bipartite matching results of Sections 3.1 and 3.3 to the assignment problem. Section 3.9 sketches the practical effects of global updates on a generic push-implementation that solves Assignment problems.

3.1 Bipartite Matching: Global Updates and the Minimum Distance Discharge Algorithm

In this section, we specify an ordering of the *push* and *relabel* operations that yields certain desirable properties. We also introduce the idea of a global distance update and show that the algorithm resulting from our operation ordering and global update strategy runs in $O(\sqrt{n}m)$ time.

For any nodes v, w, let $d_w(v)$ denote the breadth-first-search distance from v to w in the (directed) residual graph of the current preflow. If w is unreachable from v in the residual graph, $d_w(v)$ is infinite. Setting $d(v) = \min\{d_t(v), n + d_s(v)\}$ for every node $v \in V$ is called a *global update operation.* This operation also sets the current arc of every node to the node's first arc. Such an operation can be accomplished with $O(m)$ work that amounts to two breadth-first-search computations. Validity of the resulting distance labeling is a straightforward consequence of the definition.

Note that a global update cannot decrease any node's distance label, so the standard bounds on the *push* and *relabel* operations hold in the presence of global updates.

The ordering of operations we use is called *Minimum Distance Discharge*; it consists of repeatedly choosing an active node whose distance label is minimum among all active nodes and, if there is an admissible arc leaving that node, pushing a unit of flow along the admissible arc, otherwise relabeling the node. For the sake of efficient implementation and easy generalization to the weighted case, we formulate this selection strategy in a slightly different (but equivalent) way and use this formulation to guide the implementation and analysis. The intuition is that we select a unit of excess at an active node with minimum distance label, and process that unit of excess until a relabeling occurs or the excess reaches s or t. In the event of a relabeling, the new distance label may be small enough to guarantee that the same excess still has the minimum label; if so, we avoid the work associated with finding the next excess to process. This scheme's important properties generalize to the weighted case, and it allows us to show easily that the work done in active node selection is not too great.

We implement this selection rule by maintaining a collection of buckets, $b_0, \ldots, b_{2n}$; each b_i contains the active nodes with distance label i, except possibly one which is currently being processed. During the execution, we maintain λ, the index of the bucket from which we selected the most recent unit of excess. When we relabel a node, if the new distance label is no more than λ, we know that node still has minimum distance label among the active nodes, so we continue processing the same unit of excess.

In addition, we perform periodic global updates. The first global update is performed immediately after the preflow is initialized. After each *push* and *relabel* operation, the algorithm checks the following two conditions and performs a global update if both conditions hold:

- Since the most recent update, at least one unit of excess has reached s or t; and
- Since the most recent update, the algorithm has done at least m work in *push* and *relabel* operations.

Immediately after each global update, we rebuild the buckets in $O(n)$ time and set

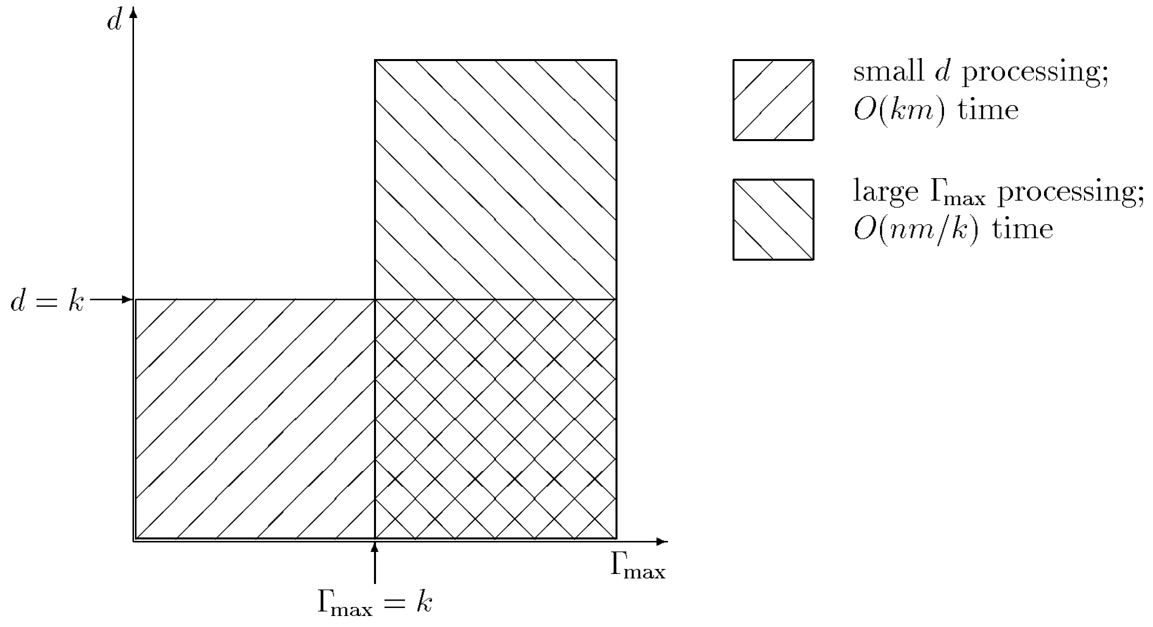

d
$d = k$
$\Gamma_{\max}$
$\Gamma_{\max} = k$
small d processing; $O(km)$ time
large $\Gamma_{\max}$ processing; $O(nm/k)$ time

Figure 3.1: Accounting for work when $0 \leq \Gamma_{\max} \leq n$

λ to zero. The following lemma shows that the algorithm does little extra work in selecting nodes to process.

Lemma 3.1.1 *Between two consecutive global updates, the algorithm does $O(n)$ work in examining empty buckets.*

Proof: Immediate, because λ decreases only when it is set to zero after an update, and there are $2n + 1 = O(n)$ buckets. ■

We will denote by $\Gamma(f, d)$ (or simply Γ) the minimum distance label of an active node with respect to the pseudoflow f and the distance labeling d. We let $\Gamma_{\max}$ denote the maximum value reached by Γ during the algorithm so far. Note that $\Gamma_{\max}$ is often equal to λ; we use the separate names mainly to emphasize that λ is maintained by the implementation, while $\Gamma_{\max}$ is an abstract quantity with relevance to the analysis regardless of the implementation details.

Figure 3.1 represents the structure underlying our analysis of the Minimum Distance Discharge algorithm. (Strictly speaking, the figure shows only half of the analysis; the part when $\Gamma_{\max} > n$ is essentially similar.) The horizontal axis corresponds to the value of $\Gamma_{\max}$ which increases as the algorithm proceeds, and the vertical axis corresponds to the distance label of the node currently being processed. Our analysis hinges on a parameter k in the range $2 \leq k \leq n$, to be chosen later. We divide the execution of the algorithm into four stages: In the first two stages, excesses are moved to t; in the final two stages, excesses that cannot reach t return to s. We analyze the first stage of each pair using the following lemma.

Lemma 3.1.2 *The Minimum Distance Discharge algorithm expends $O(km)$ work during the periods when $\Gamma_{\max} \in [0, k]$ and $\Gamma_{\max} \in [n, n+k]$.*

Proof: First, note that if $\Gamma_{\max}$ falls in the first interval of interest, Γ must lie in that interval as well. This relationship also holds for the second interval after a global update is performed. Since the work from the beginning of the second interval until the price update is performed is $O(m)$, it is enough to show that the time spent by the algorithm during periods when $\Gamma \in [0, k]$ and $\Gamma \in [n, n+k]$ is in $O(km)$. Note that the periods defined in terms of Γ may not represent contiguous intervals during the execution of the algorithm.

Each node can be relabeled at most $k+1$ times when $\Gamma \in [0, k]$, and similarly for $\Gamma \in [n, n+k]$. Hence the relabelings and pushes require $O(km)$ work. The observations that a global update requires $O(m)$ work and during each period there are $O(k)$ global updates complete the proof. ■

To study the behavior of the algorithm during the remainder of its execution, we exploit the structure of matching networks by appealing to a combinatorial lemma. The following lemma is a special case of a well-known decomposition theorem [20] (see also [18]). The proof depends mainly on the fact that for a matching network G, the in-degree of $v \in X$ in G_f is $1 - e_f(v)$ and the out-degree of $w \in Y$ in G_f is $1 + e_f(w)$ for any integral pseudoflow f.

Lemma 3.1.3 *Any integral pseudoflow f in the residual graph of an integral flow g in a matching network can be decomposed into cycles and simple paths that are pairwise node-disjoint except at the endpoints of the paths, such that each element in* *the decomposition carries one unit of flow. Each path is from a node v with $e_f(v) < 0$ (v can be t) to a node w with $e_f(w) > 0$ (w can be s).*

Lemma 3.1.3 allows us to show that when $\Gamma_{\max}$ is outside the intervals covered by Lemma 3.1.2, the amount of excess the algorithm must process is small.

Given a preflow f, we define the *residual flow value* to be the total excess that can reach t in G_f.

Lemma 3.1.4 *If $\Gamma_{\max} \geq k > 2$, the residual flow value is at most $n/(k-1)$ if G is a matching network.*

Proof: Note that the residual flow value never increases during an execution of the algorithm, and consider the pair (f, d) such that $\Gamma(f, d) \geq k$ for the first time during the execution. Let f^* be a maximum flow in G, and let $f' = f^* - f$. Now $-f'$ is a pseudoflow in G_{f^*}, and therefore can be decomposed into cycles and paths as in Lemma 3.1.3. Such a decomposition of $-f'$ induces the obvious decomposition on f' with all the paths and cycles reversed and excesses negated. Because $\Gamma \geq k$ and d is a valid distance labeling with respect to f, any path in G_f from an active node to t must contain at least $k + 1$ nodes. In particular, the excess-to-t paths in the decomposition of f' contain at least $k + 1$ nodes each, and are node-disjoint except for their endpoints. Since G contains only $n + 2$ nodes, there can be no more than $n/(k - 1)$ such paths. Since f^* is a maximum flow, the amount of excess that can reach t in G_f is no more than $n/(k - 1)$. ■

The proof of the next lemma is similar.

Lemma 3.1.5 *If $\Gamma_{\max} \geq n+k > n+2$ during an execution of the Minimum Distance Discharge algorithm with global updates on a matching network, the total excess at nodes in V is at most $n/(k-1)$.*

The following lemma shows an important property of the rules we use to trigger global update operations, namely that during a period when the algorithm does $\Theta(m)$ work at least one unit of excess is guaranteed to reach s or t.

Lemma 3.1.6 *Between any two consecutive global update operations, the algorithm does* $\Theta(m)$ *work.*

Proof: According to the two conditions that trigger a global update, it suffices to show that immediately after an update, the work done in moving a unit of excess to s or t is $O(m)$. For every node v, at least one of $d_s(v)$, $d_t(v)$ is finite. Therefore, immediately after a global update, at least one admissible arc leaves every node except s and t, by definition of the global update operation. Recall that the admissible graph is acyclic, so the first unit of excess processed by the algorithm immediately after a global update arrives at t or at s before any relabeling occurs, and does so along a simple path. Consider the path taken by the flow unit to s or t. The work performed while moving the unit along the path is proportional to the length of the path plus the number of times current arcs of nodes on the path are advanced. This $O(n+m) = O(m)$ work is performed before the the first condition for a global update is met.

Following an amount of additional work bounded above by $m + O(n)$, plus work proportional to that for a *push* or *relabel* operation, another global update operation will be triggered. Clearly a *push* or *relabel* takes $O(m)$ work and the lemma follows. ■

We are ready to prove the main result of this section.

Theorem 3.1.7 *The Minimum Distance Discharge algorithm with global updates computes a maximum flow in a matching network (and hence a maximum cardinality bipartite matching) in* $O(\sqrt{n}m)$ *time.*

Proof: By Lemma 3.1.2, the total work done by the algorithm when $\Gamma_{\max} \in [0, k]$ and $\Gamma_{\max} \in [n, n+k]$ is $O(km)$. By Lemmas 3.1.4 and 3.1.5, the amount of excess processed when $\Gamma_{\max}$ falls outside these bounds is at most $2n/(k-1)$. From Lemma 3.1.6 we conclude that the work done in processing this excess is $O(nm/k)$. Hence the time bound for the Minimum Distance Discharge algorithm is $O(km + nm/k)$. Choosing $k = \Theta(\sqrt{n}\,)$ to balance the two terms, we see that the Minimum Distance Discharge algorithm with global updates runs in $O(\sqrt{n}m)$ time. ■

3.2 Improved Performance through Graph Compression

Feder and Motwani [19] give an algorithm that runs in $o(\sqrt{n}m)$ time and produces a *compressed representation* $\overline{G}^* = (\overline{V} \cup W, \overline{E}^*)$ of a bipartite graph in which all adjacency information is preserved, but that has asymptotically fewer edges if the original graph $\overline{G} = (\overline{V}, \overline{E})$ is dense. This graph consists of all the original nodes of X and Y, as well as a set of additional nodes W. An edge $\{x, y\}$ appears in $\overline{E}$ if and only if either $\{x, y\} \in \overline{E}^*$ or $\overline{G}^*$ contains a length-two path from x to y through some node of W.

The following theorem is slightly specialized from Feder and Motwani's Theorem 3.1 [19], which details the performance of their algorithm *Compress*:

Theorem 3.2.1 *Let $\delta \in (0,1)$ and let $\overline{G} = (\overline{V} = X \cup Y, \overline{E})$ be an undirected bipartite graph with $|X| = |Y| = n$ and $|\overline{E}| = m \geq n^{2-\delta}$. Then algorithm* Compress *computes a compressed representation $\overline{G}^* = (\overline{V} \cup W, \overline{E}^*)$ of $\overline{G}$ with $m^* = |\overline{E}^*| = O\left(m\delta^{-1}\frac{\log(n^2/m)}{\log n}\right)$ in time $O(mn^\delta \log^2 n)$. The number of nodes in W is $O(mn^{\delta-1})$.*

In particular, we choose a constant $\delta < 1/2$; then the compressed representation is computed in time $o(\sqrt{n}m)$ and has $m^* = O\left(m\frac{\log(n^2/m)}{\log n}\right)$ edges.

Given a compressed representation $\overline{G}^*$ of $\overline{G}$, we can compute a flow network G^* in which there is a correspondence between flows in G^* and matchings in $\overline{G}$. The only differences from the reduction of Section 2.1.1 are that each edge $\{x, w\}$ with $x \in X$ and $w \in W$ gives an arc (x, w), and each edge $\{w, y\}$ with $w \in W$ and $y \in Y$ gives an arc (w, y). As in Section 2.1.1, we have a relationship between matchings in the original graph $\overline{G}$ and flows in G^*, but now the correspondence is not one-to-one as it was before. Nevertheless, it remains true here that given a flow f with $e_f(t) = c$ in G^*, we can find a matching of cardinality c in $\overline{G}$ using only $O(n)$ time in a straightforward way.

The performance improvement we gain comes by using the graph compression step as preprocessing: we will show that the Minimum Distance Discharge algorithm with global updates runs in time $O(\sqrt{n}m^*)$ on the flow network G^* corresponding to the compressed representation $\overline{G}^*$ of a bipartite graph $\overline{G}$. In other words, the speedup results only from the reduced number of edges, not from changes within the Minimum Distance Discharge algorithm.

To prove the performance bound, we must generalize certain lemmas from Section 3.1 to networks with the structure of compressed representations. Lemma 3.1.2 is independent of the input network's structure, as are Lemma 3.1.6 and Lemma 3.1.1. An analogue to Lemma 3.1.3 holds in a flow network derived from a compressed representation; this will extend Lemmas 3.1.4 and 3.1.5, allowing us to conclude the improved time bound.

Lemma 3.2.2 *Any integral pseudoflow f in the residual graph of an integral flow g in the flow graph of a compressed representation can be decomposed into cycles and simple paths that are pairwise node-disjoint at nodes of X and Y except at the endpoints of the paths, such that each element of the decomposition carries one unit of flow. Each path is from a node v with $e_f(v) < 0$ (v can be t) to a node w with $e_f(w) > 0$ (w can be s).*

Proof: As with matching networks, the in-degree of $v \in X$ is $1 - e_f(v)$ and the out-degree of $y \in Y$ is $1 + e_f(y)$, so the standard proof of Lemma 3.1.3 extends to this case. ■

The following lemma is analogous to Lemma 3.1.4.

Lemma 3.2.3 *If $\Gamma_{\max} \geq k > 2$, the residual flow value is at most $2n/(k-2)$ if G^* is a compressed representation.*

Proof: As in the case of Lemma 3.1.4, except that here an excess-to-t path in the decomposition of f' must contain at least $k/2$ nodes of $\overline{V}$. Since $\overline{V}$ contains only n nodes, there can be no more than $2n/(k-2)$ such paths, and so because f^* is a maximum flow, the amount of excess that can reach t in G^*_f is no more than $2n/(k-2)$. ■

The following lemma is analogous to Lemma 3.1.5, and its proof is similar to the proof of Lemma 3.2.3.

Lemma 3.2.4 *If* $\Gamma_{\max} \geq n+k > n+2$ *during an execution of the Minimum Distance Discharge algorithm with global updates on a compressed representation, the total excess at nodes in* $\overline{V} \cup W$ *is at most* $2n/(k-2)$.

Using the same reasoning as in Theorem 3.1.7, we have:

Theorem 3.2.5 *The Minimum Distance Discharge algorithm with global updates computes a maximum flow in the network corresponding to a compressed representation with* m^* *edges in* $O(\sqrt{n}m^*)$ *time.*

To complete our time bound for the bipartite matching problem we must dispense with some technical restrictions in Theorem 3.2.1, namely the requirements that $|X| = |Y| = n$ and that $m \geq n^{2-\delta}$. The former condition is easily met by adding nodes to whichever of X, Y is the smaller set, so their cardinalities are equal. These "dummy" nodes are incident to no edges. As for the remaining condition, observe that our time bound does not suffer if we simply forego the compression step and apply the result of Section 3.1 in the case where $m < n^{2-\delta}$. To see this, recall that we chose $\delta < 1/2$, and note that $1 \leq m < n^{2-\delta}$ implies $\frac{\log(n^2/m)}{\log n} = \Theta(1)$. So we have:

Theorem 3.2.6 *The Minimum Distance Discharge algorithm with graph compression and global updates computes a maximum cardinality bipartite matching in time* $O\left(\sqrt{n}m\frac{\log(n^2/m)}{\log n}\right)$.

This bound matches that of Feder and Motwani for Dinitz's algorithm.

3.3 Unweighted Bad Example: Minimum Distance Discharge Algorithm without Global Updates

In this section we describe a family of graphs on which the Minimum Distance Discharge algorithm *without* global updates requires $\Omega(nm)$ time (for values of m between $\Theta(n)$ and $\Theta(n^2)$). This shows that the updates improve the worst-case running time of the algorithm. The goal of our construction is to exhibit an execution of the algorithm in which each relabeling changes a node's distance label by $O(1)$. Under this condition the execution will have to perform $\Omega(n^2)$ relabelings, and these relabelings will require $\Omega(nm)$ time.

Given $\tilde{n} \in \mathbf{Z}$ and $\tilde{m} \in [1, \tilde{n}^2/4]$, we construct a graph $\overline{G}$ as follows: $\overline{G}$ is the complete bipartite graph with $\overline{V} = X \cup Y$, where

$$X = \left\{ 1, 2, \ldots, \left\lceil \frac{\tilde{n} + \sqrt{\tilde{n}^2 - 4\tilde{m}}}{2} \right\rceil \right\} \qquad \text{and} \qquad Y = \left\{ 1, 2, \ldots, \left\lfloor \frac{\tilde{n} - \sqrt{\tilde{n}^2 - 4\tilde{m}}}{2} \right\rfloor \right\}.$$

It is straightforward to check that this graph has $n = \tilde{n} + O(1)$ nodes and $m = \tilde{m} + O(\tilde{n})$ edges. Note that $|X| > |Y|$.

Figure 3.2 describes a particular execution of the Minimum Distance Discharge algorithm on G, the matching network derived from $\overline{G}$, that requires $\Omega(nm)$ time. With more complicated but unilluminating analysis, it is possible to show that every execution of the Minimum Distance Discharge algorithm on G requires $\Omega(nm)$ time.

It is straightforward to verify that in the execution outlined, all processing takes place at active nodes whose distance labels are minimum among the active nodes. The algorithm performs poorly because during the execution, no relabeling changes a distance label by more than two. Hence the execution uses $\Theta(nm)$ work in the course of its $\Theta(n^2)$ relabelings, and we have the following theorem:

Theorem 3.3.1 *For any function $m(n)$ in the range $n \leq m(n) < n^2/4$, there exists an infinite family of instances of the bipartite matching problem having $\Theta(n)$ nodes and $\Theta(m(n))$ edges on which the Minimum Distance Discharge algorithm without global updates runs in $\Omega(nm(n))$ time.*

3.4 Unweighted Global Updates in Practice: Effects on a Generic Implementation

Bipartite matching is a relatively easy subclass of maximum flow problems, and implementation studies have traditionally not investigated codes' performance on this subclass. Nevertheless, it is interesting for us to study the effects of the the global

```
1. Initialization establishes $|X|$ units of excess, one at each node of $X$;
2. Nodes of $X$ are relabeled one-by-one, so all $v \in X$ have $d(v) = 1$;
3. While $e_f(t) < |Y|$,
    3.1. a unit of excess moves from some node $v \in X$ to some node $w \in Y$ with $d(w) = 0$;
    3.2. $w$ is relabeled so that $d(w) = 1$;
    3.3. The unit of excess moves from $w$ to $t$, increasing $e_f(t)$ by one.
4. A single node, $x_1$ with $e_f(x_1) = 1$, is relabeled so that $d(x_1) = 2$.
5. $\ell \leftarrow 1$.
6. While $\ell \leq n$,
        Remark: All nodes $v \in V$ now have $d(v) = \ell$ with the exception of the one node
        $x_\ell \in X$, which has $d(x_\ell) = \ell + 1$ and $e_f(x_\ell) = 1$; all excesses are at nodes of $X$;
    6.1. All nodes with excess, except the single node $x_\ell$, are relabeled one-by-one so
         that all $v \in X$ with $e_f(v) = 1$ have $d(v) = \ell + 1$;
    6.2. While some node $y \in Y$ has $d(y) = \ell$,
        6.2.1. A unit of excess is pushed from a node in $X$ to $y$;
        6.2.2. $y$ is relabeled so $d(y) = \ell + 1$;
        6.2.3. The unit of excess at $y$ is pushed to a node $x \in X$ with $d(x) = \ell$;
        6.2.4. $x$ is relabeled so that if some node in $Y$ still has distance label $\ell$,
                   $d(x) = \ell + 1$;
                  otherwise
                   $d(x) = \ell + 2$ and $x_{\ell+1} \leftarrow x$;
    6.3. $\ell \leftarrow \ell + 1$;
7. Excesses are pushed one-by-one from nodes in $X$ (labeled $n + 1$) to $s$.
```

Figure 3.2: The Minimum Distance Discharge execution on bad examples.

update heuristic in the context where the theoretical results of Section 3.1 apply. Therefore, we describe a brief series of experiments we conducted to examine the effects of global updates on a modified version of an implementation developed in [9]. The implementation that served as our starting point is called M_PRF [9]; it uses periodic global updates and a maximum-distance node selection strategy. See [9] for a detailed description of M_PRF.

We began our study by modifying the M_PRF implementation in the following ways:

- Minimum distance active node selection was substituted for maximum distance; and
- The global update heuristic was made switchable, so the code's performance with and without global updates could be compared.

We used minimum distance selection for consistency with the theoretical analysis; spot-checks on several instances suggest that the selection strategies of [9] perform about the same as minimum distance selection on bipartite matching problems. To keep the code's bookkeeping simple, we kept the default frequency of global updates: the implementation that used global updates performed one global update operation after every n relabelings. This number of relabelings will generally require $\Theta(m)$ work; an implementation that strictly enforced the condition that $\Theta(m)$ work is done between global updates would perform very similarly to the one we used.

3.4.1 Problem Classes and Experiment Outline

We compared minimum distance selection with and without global updates on the problem classes described below. The codes' running times may be sensitive to the order in which the graph edges are listed in the input and so to suppress such artifacts in the running time, we applied a pseudorandom permutation to the edges of each graph before supplying it to the max-flow codes.

Worst-Case problems

Matching problems in this class are the ones described in Section 3.3, with $|X| = |Y| + 1$. These problems are called worst-case problems because they elicit worst-case running time for the code without global updates. The only difference between instances of the same size is the permutation applied to the edges.

Long Path problems

The edges in long path problems form a single long path. Let $X = \{1, \ldots, n/2\}$ and $Y = \{n/2 + 1, \ldots, n\}$. For $1 < i \leq n/2$, node i has edges to nodes $n/2 + i$ and $n/2 + i - 1$. Node 1 has an edge to node $n/2 + 1$. The graph structure is completely determined by the problem size in this class; only the edge permutation varies.

Very Sparse problems

Problems in this class have $m = n/2$ edges, each between a pair of nodes chosen uniformly at random from the set $X \times Y$. As problem size grows, instances that admit a perfect matching become exceedingly rare in this class. The graph structure varies between different instances of the same size in this problem class.

Unique-Dense problems

Problems in this class are dense (*i.e.*, $m = \Theta(n^2)$), but admit only one perfect matching. Let $X = \{1, \ldots, n/2\}$ and $Y = \{n/2+1, \ldots, n\}$. Then node i has edges to nodes $n/2+j$ for all $1 \le j \le i$. Instances of a particular size in this class differ only in their edge permutations.

3.4.2 Running Times and Discussion

The maximum flow codes were compiled using `gcc` version 2.6.3 with the `-O2` optimization switch, and were run on a 40-MHz SUN Sparc-10 processor with 160 megabytes of main memory under SunOS version 4.1.3. We report average running times and standard deviations computed over three problem instances for each size and class. All times are given in seconds.

Worst-Case problems

We experimented on this class to give concrete reinforcement to the analysis of Section 3.1. In accordance with the analysis, one would expect the code without global

	No GU		With GU		
n	time	s	time	s	speedup
201	0.9	1e-8	0.02	3e-10	47
1001	179	0.2	0.6	0.01	301
2001	1591	0.46	2.6	0.01	613

Figure 3.3: Running time comparison on the Worst-Case class

updates to perform poorly on this problem class, and indeed it does. Problems in this class are trivial for the code that uses global updates, and so in this somewhat contrived context global updates improve performance by a factor of several hundred even for small instances. The factor of improvement increases with problem size.

Long Path problems

	No GU		With GU		
n	time	s	time	s	speedup
10000	1.7	0.4	0.76	0.12	2.3
100000	69	16	25	2.0	2.8

Figure 3.4: Running time comparison on the Long Path class

Global updates improve performance on this class by a moderate factor that appears to grow slightly with problem size.

Very Sparse problems

	No GU		With GU		
n	time	s	time	s	speedup
10000	67.9	4.5	0.22	0.02	309
100000	9965	208	4.1	0.1	2430

Figure 3.5: Running time comparison on the Very Sparse class

The running time of the code without global updates is huge relative to the time used by the code with global updates, seemingly because the algorithm without global updates has a difficult time discovering that some unit(s) of excess must be returned to the source. This is essentially the same phenomenon as brought out by the worst-case class.

Unique-Dense problems

	No GU		With GU		
n	time	s	time	s	speedup
500	0.68	0.1	0.33	0.04	2.0
1000	5.5	0.2	2.4	0.07	2.3
2000	45	4	13.0	0.3	3.4

Figure 3.6: Running time comparison on the Unique-Dense class

On this class as on the long path class, global updates account for an improvement in running time by a factor that appears to be roughly a relatively small but substantial constant.

To summarize, on every graph family we studied, global updates improved the running time of the implementation by a significant (sometimes huge) amount. Moreover, the updates make the code much more robust: without global updates, the running time variance on some families is huge; with them, the variance is consistently small.

Although this brief study is far from exhaustive, our data show that global updates can provide a substantial overall performance gain in a maximum flow implementation applied to bipartite matching problems.

3.5 Assignment Problem: Global Updates and the Minimum Change Discharge Algorithm

In this section, we generalize the ideas of minimum distance discharge and global updates to the context of the minimum cost circulation problem and analyze the algorithm that embodies these generalizations.

We analyze a single execution of *refine*, and to simplify our notation, we make some assumptions that do not affect the results. We assume that the price function is identically zero at the beginning of the iteration. Our analysis goes through without this assumption, but the required condition can be achieved at no increased asymptotic cost by replacing the arc costs with their reduced costs and setting the node prices to zero in the first step of *refine*.

Under the assumption that each iteration begins with the zero price function, the *price change* of a node v during an iteration is $\delta(v) = \lfloor -p(v)/\epsilon \rfloor$. By analogy to the matching case, we define $\Gamma(f, p) = \min_{e_f(v)>0}\{\delta(v)\}$, and let $\Gamma_{\max}$ denote the maximum value attained by $\Gamma(f, p)$ so far in this iteration. The *minimum change discharge* strategy consists of repeatedly selecting a unit of excess at an active node v with $\delta(v) = \Gamma$ and processing that unit until it cancels some deficit or a relabeling occurs. We implement this strategy as in the unweighted case. Observe that no active node's price changes by more than $2\alpha n\epsilon$ during refine, so a collection of $2\alpha n+1$ buckets $b_0, \ldots, b_{2\alpha n}$ is sufficient to keep every active node v in $b_{\delta(v)}$. As before, the algorithm maintains the index λ of the lowest-numbered nonempty bucket and avoids bucket access except immediately after a deficit is canceled or a relabeling of a node v sets $\delta(v) > \lambda$.

In the weighted context, a global update takes the form of setting each node price so that G_A is acyclic, there is a path in G_A from every excess to some deficit (a node v with $e_f(v) < 0$) and every node reachable in G_A from a node with excess lies on such a path. This amounts to a modified shortest-paths computation, and can be done in $O(m)$ time using ideas from Dial's work [15]. We specify the global update algorithm in detail and prove the required properties in Section 3.6. At every *refine*, the first global update is performed immediately after saturating all residual arcs with negative reduced cost. After each *push* and *relabel* operation, the algorithm checks the following two conditions and performs a global update if both conditions hold:

- Since the most recent update, at least one unit of excess has canceled some deficit; and
- Since the most recent update, the algorithm has done at least m work in *push* and *relabel* operations.

We developed global updates from an implementation heuristic for the minimum cost circulation problem [25], but in retrospect they prove similar in the assignment

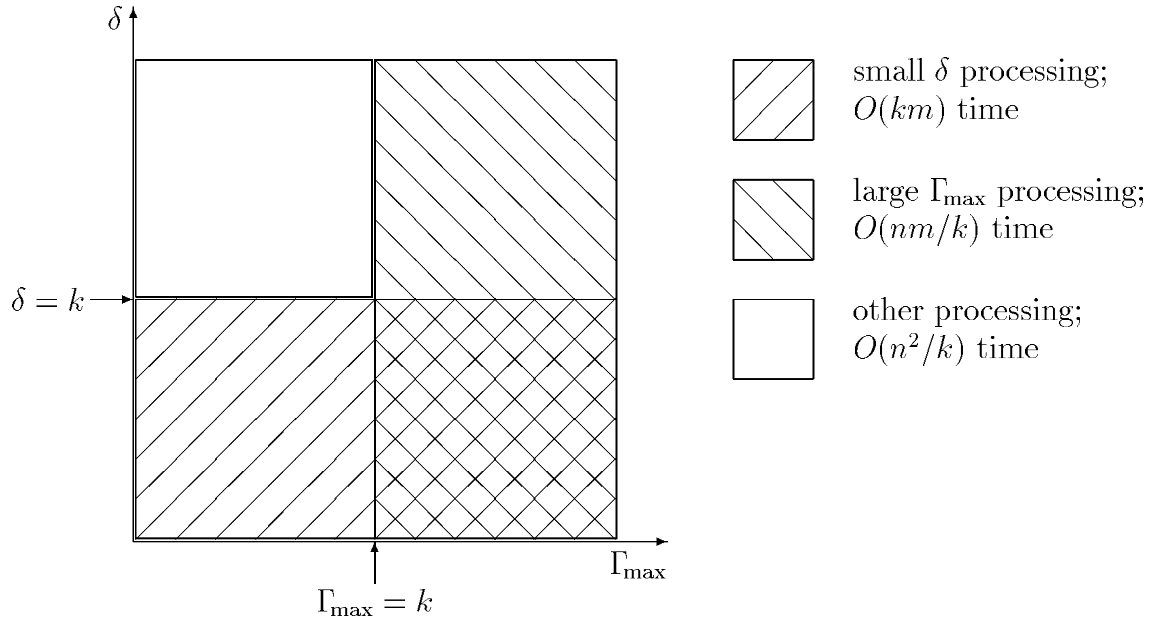

δ
$\delta = k$
$\Gamma_{\max}$
$\Gamma_{\max} = k$
small δ processing; $O(km)$ time
large $\Gamma_{\max}$ processing; $O(nm/k)$ time
other processing; $O(n^2/k)$ time

Figure 3.7: Accounting for work in the Minimum Change Discharge algorithm

context to the one-processor Hungarian Search technique developed in [22].

Immediately after each global update, the algorithm rebuilds the buckets in $O(n)$ time and sets λ to zero. As in the unweighted case, we have the following easy bound on the extra work done by the algorithm in selecting nodes to process:

Lemma 3.5.1 *Between two consecutive global updates, the algorithm does $O(n)$ work in examining empty buckets.*

Figure 3.7 represents the main ideas behind our analysis of an iteration of the Minimum Change Discharge algorithm. The diagram differs from Figure 3.1 because we must account for pushes and relabelings that occur at nodes with large values of δ when $\Gamma_{\max}$ is small. Such operations could not arise in the matching algorithm, but are possible here.

We begin our analysis with a lemma that is essentially similar to Lemma 3.1.2.

Lemma 3.5.2 *The algorithm does $O(km)$ work in the course of* relabel *operations on nodes v obeying $\delta(v) \leq k$ and* push *operations from those nodes.*

Proof: A node v can be relabeled at most $k+1$ times while $\delta(v) \leq k$, so the relabelings of such nodes and the pushes from them require $O(km)$ work. ∎

To analyze our algorithm for the assignment problem, we must overcome two main difficulties that were not present in the matching case. First, we can do *push* and *relabel* operations at nodes whose price changes are large even when $\Gamma_{\max}$ is small; this work is not bounded by Lemma 3.5.2 and we must account for it. Second, our analysis of the period when $\Gamma_{\max}$ is large in the unweighted case does not generalize because it is not true that $\delta(v)$ gives a bound on the breadth-first-search distance from v to a deficit in the residual graph.

Lemma 3.5.5 is crucial in resolving both of these issues, and to prove it we use the following standard result which is analogous to Lemma 3.1.3.

Lemma 3.5.3 *Given a matching network G and an integral circulation g, any integral pseudoflow f in G_g can be decomposed into*

- *cycles and*
- *paths, each from a node u with $e_f(u) < 0$ to a node v with $e_f(v) > 0$,*

where all the elements of the decomposition are pairwise node-disjoint except at s, t, and the endpoints of the paths, and each element carries one unit of flow.

We denote a path from node u to node v in such a decomposition by $(u \rightsquigarrow v)$.

The next lemma makes clear how our analysis benefits from the asymmetric definitions of ϵ-optimality and admissibility.

Lemma 3.5.4 *For any value of $\epsilon \geq 0$, there are at most n arcs with negative reduced cost in the residual graph of any integral ϵ-optimal circulation.*

Proof: It is straightforward to verify that for any matching network G and integral circulation g, the residual graph G_g has exactly n arcs $a \notin E_U$, and so the lemma follows directly from the asymmetric definition of ϵ-optimality. ∎

The following lemma is similar in spirit to those in [22] and [30], although the single-phase push-relabel framework of our algorithm changes the structure of the proof. Let $\mathcal{E}(f)$ denote the total excess in pseudoflow f, *i.e.*, $\sum_{e_f(v)>0} e_f(v)$. When no confusion will arise, we simply use $\mathcal{E}$ to denote the total excess in the current pseudoflow. The lemma depends on the $(\alpha\epsilon)$-optimality of the circulation produced by the previous iteration of *refine*, so it holds only in the second and subsequent scaling iterations. Because the zero circulation is not C-optimal with respect to the zero price function, we need different phrasing to accomplish the same task in the first iteration. The differences are mainly technical, so the first-iteration lemmas and their proofs are postponed to Section 3.8.

Lemma 3.5.5 *At any point during an execution of* refine *other than the first,* $\mathcal{E} \times \Gamma_{\max} \leq 2((5+\alpha)n-1)$.

Proof: Let c denote the (reduced) arc cost function at the beginning of this execution of *refine*, and let $G = (V, E)$ denote the residual graph at the same instant. For simplicity in the following analysis, we view a pseudoflow as an entity in this graph G. Let f, p be the current pseudoflow and price function at the most recent point during the execution of *refine* when $\Gamma(f,p) = \Gamma_{\max}$. Then we have

$$\mathcal{E}(f) \times \Gamma_{\max} \leq \sum_{e_f(v)>0} \delta(v) e_f(v).$$

From the definition of δ, then,

$$\mathcal{E}(f) \times \Gamma_{\max} \times \epsilon \leq -\sum_{e_f(v)>0} p(v) e_f(v).$$

We will complete our proof by showing that

$$-\sum_{e_f(v)>0} p(v) e_f(v) = c_p(f) - c(f)$$

and then deriving an upper bound on this quantity.

By the definition of the reduced costs,

$$c_p(f) - c(f) = \sum_{f(v,w)>0} (p(v) - p(w)) f(v, w).$$

Letting $\mathcal{P}$ be a decomposition of f into paths and cycles according to Lemma 3.5.3 and noting that cycles make no contribution to the sum, we can rewrite this expression as

$$\sum_{(u \leadsto v) \in \mathcal{P}} (p(u) - p(v)).$$

Since nodes u with $e_f(u) < 0$ are never relabeled, $p(u) = 0$ for such a node, and we have

$$c_p(f) - c(f) = -\sum_{(u \leadsto v) \in \mathcal{P}} p(v).$$

Because the decomposition $\mathcal{P}$ must account for all of f's excesses and deficits, we can rewrite

$$c_p(f) - c(f) = -\sum_{e_f(v) > 0} p(v) e_f(v).$$

Now we derive an upper bound on $c_p(f) - c(f)$. By Lemma 3.5.4, there are at most n negative-cost arcs in E, and each has cost at least $-2\alpha\epsilon$ because *refine* begins with the residual graph of an $(\alpha\epsilon)$-optimal circulation. Therefore we have $c(f) \geq -2\alpha n\epsilon$ and hence $c_p(f) - c(f) \leq c_p(f) + 2\alpha n\epsilon$.

Now consider $c_p(f)$. Clearly, $f(a) > 0 \implies a^R \in E_f$, and ϵ-optimality of f with respect to p says that $a^R \in E_f \implies c_p(a^R) \geq -2\epsilon$. Since $c_p(a^R) = -c_p(a)$, we have $f(a) > 0 \implies c_p(a) \leq 2\epsilon$. Recalling our decomposition $\mathcal{P}$ into cycles and paths from deficits to excesses, observe that $c_p(f) = \sum_{P \in \mathcal{P}} c_p(P)$. Let $\nu(P)$ denote the interior of a path P, *i.e.*, the path minus its endpoints and initial and final arcs, and let $\partial(P)$ denote the set containing the initial and final arcs of P. If P is a cycle, $\nu(P) = P$ and $\partial(P) = \emptyset$. We can write

$$c_p(f) = \sum_{P \in \mathcal{P}} c_p(\nu(P)) + \sum_{P \in \mathcal{P}} c_p(\partial(P)).$$

The total number of arcs not incident to s or t in the cycles and path interiors is at most n by node-disjointness, and the number of arcs incident to s or t is at most $2n-1$. Also, the total excess is never more than n by Lemma 3.5.4, so the initial and final arcs of the paths number no more than $2n$. And because each arc carrying positive flow has reduced cost at most 2ϵ, we have $c_p(f) \leq (n + 2n - 1 + 2n)2\epsilon = (5n-1)2\epsilon$.

Therefore, $c_p(f)-c(f) \leq 2((5+\alpha)n-1)\epsilon$, and we have $\mathcal{E}(f)\times\Gamma_{\max} \leq 2((5+\alpha)n-1)$. ■

Corollary 3.5.6 *$\Gamma_{\max} \geq k$ implies $\mathcal{E} = O(n/k)$.*

We could use the symmetric definition of ϵ-optimality for much of the proof of Lemma 3.5.5 by substituting a slightly different measure of progress for $c_p(f) - c(f)$, but the asymmetric definition is crucial in establishing the vital fact that there are never more than n units of excess.

We use the following lemma to show that when $\Gamma_{\max}$ is small, we do a limited amount of work at nodes whose price changes are large.

Lemma 3.5.7 *While $\Gamma_{\max} \leq k$, the amount of work done in relabelings at nodes v with $\delta(v) > k$ and pushes from those nodes is $O(n^2/k)$.*

Proof: For convenience, we say a node that gets relabeled under the conditions of the lemma is a *bad* node. We process a given node v either because we selected a unit of excess at v, or because the most recent operation was a *push* from one of v's neighbors to v. If a unit of v's excess is selected, we have $\delta(v) \leq \Gamma_{\max}$ (indeed without global updates, $\delta(v) = \Gamma_{\max}$) which implies $\delta(v) \leq k$, so v cannot be a bad node. In the second case, the unit of excess just pushed to v will remain at v until $\Gamma_{\max} \geq \delta(v)$ because the condition $\delta(v) > \lambda$ will cause excess at a different node to be selected immediately after v is relabeled. We cannot select v's excess until $\Gamma_{\max} \geq \delta(v)$, and at such a time, Corollary 3.5.6 shows that the total excess remaining is $O(n/k)$. Since each relabeling of a bad node leaves a unit of excess that must remain at that node until $\Gamma_{\max} \geq k$, the number of relabelings of bad nodes is $O(n/k)$. Because every node has degree at most n, the work done in pushes and relabelings at bad nodes is $O(n^2/k)$. ■

Recall that the algorithm initiates global update only after a unit of excess has canceled some deficit since the last global update. The next lemma, analogous to Lemma 3.1.6, shows that this rule cannot introduce too great a delay.

Lemma 3.5.8 *Between any two consecutive global update operations, the algorithm does $\Theta(m)$ work.*

Proof: As in the unweighted case, it suffices to show that the algorithm does $O(m)$ work in canceling a deficit immediately after a global update operation, and $O(m)$ work in selecting nodes to process. Theorems 3.6.7 and 3.6.8 (see Section 3.6) suffice to ensure that a unit of excess reaches some deficit immediately after a global update and before any relabeling occurs, and Lemma 3.5.1 shows that the extra work done between global updates in selecting nodes to process is $O(n)$. ∎

Lemmas 3.5.2 and 3.5.7, along with Theorem 3.6.9 show that the algorithm takes $O(km+n^2/k)$ time when $\Gamma_{\max} \leq k$. Corollary 3.5.6 says that when $\Gamma_{\max} \geq k$, the total excess remaining is $O(n/k)$, and Lemma 3.5.8 together with Theorem 3.6.9 shows that $O(m)$ work suffices to cancel each unit of excess remaining. Therefore the total work in an execution of *refine* is $O(km + n^2/k + nm/k)$, and choosing $k = \Theta(\sqrt{n})$ gives a $O(\sqrt{n}m)$ time bound on an execution of *refine*. The overall time bound follows from the $O(\log(nC))$ bound on the number of scaling iterations, giving the following theorem:

Theorem 3.5.9 *The Minimum Change Discharge algorithm with global updates computes a minimum cost circulation in a matching network in* $O(\sqrt{n}m\log(nC))$ *time.*

Graph compression methods [19] do not apply to graphs with weights because the compressed graph preserves only adjacency information and cannot encode arbitrary edge weights. Hence the Feder-Motwani techniques by themselves cannot improve performance in the assignment problem context.

3.6 Weighted Global Updates: a Variant of Dial's Algorithm

Now we develop the structure of global updates in the weighted case, and prove the properties required for the analysis given in Section 3.5.

3.6.1 The Global Update Algorithm: Approximate Shortest Paths

A global update operation amounts to a modified form of Dijkstra's algorithm for shortest paths, implemented using Dial's idea that applies when the range of distances is small and the arc lengths are integers [15]. For the purposes of global updates, we define the *length* ℓ of arc a as follows:

$$\ell(a) = \begin{cases} \lfloor c_p(a)/\epsilon \rfloor & \text{if } a \in E_U; \\ 1 + \lfloor c_p(a)/\epsilon \rfloor & \text{otherwise.} \end{cases}$$

Intuitively $\ell(v, w)$ is the amount, in units of ϵ, by which the quantity $p(v) - p(w)$ would have to decrease to make (v, w) an admissible arc.

Let D be the set of nodes with deficit (*i.e.*, the set of nodes w with $e_f(w) < 0$) at the time of a global update. Define $d_\ell(v)$ to be the distance of node v from the set D in G_f with respect to the length function ℓ, and imagine for a moment that we calculate $d_\ell(v)$ for every node v. If we reduce the price of every node v by $d_\ell(v) \times \epsilon$, we will preserve ϵ-optimality and acyclicity of the admissible graph, guarantee that there is an admissible path from every excess to a deficit, and guarantee that there are no "dead-ends" in the admissible graph. But we would like our global updates to run in linear time, and we know of no way to calculate d_ℓ for the entire graph so efficiently since we cannot bound the size of $d_\ell(v)$ for all nodes v. Nevertheless, we do have a bound on the price change of any *active* node during the execution of refine, and such a bound immediately limits the range of values d_ℓ can take on a node that is active at any future time during the present execution of *refine*. In particular, we have the following lemma (essentially the same as Lemma 5.8 from [32], with a slightly looser bound because our initial circulation is not C-optimal in the first iteration):

Lemma 3.6.1 *Let v have $e_f(v) > 0$, and suppose f is ϵ-optimal. Then the amount by which $p(v)$ has changed during this execution of* refine *is less than $2n\alpha\epsilon$.*

The proof of Lemma 3.6.1 holds whether or not global updates are used; it depends only on the approximate optimality properties of the current pseudoflow and the present iteration's initial circulation.

```
procedure GLOBAL-UPDATE($f, p$);
    INITIALIZE-PRIORITY-QUEUE($Q$);
    $\forall v \in D,\ \kappa[v] \leftarrow 0$;
    $\forall v \notin D,\ \kappa[v] \leftarrow 2\alpha n$;
    for each $v$ in the network do
        INSERT($Q, v, \kappa[v]$);
    $\mathit{last\text{-}key} \leftarrow 0$;
    $\mathit{total\text{-}d} \leftarrow -\sum_{v \in D} e_f(v)$;
    $\mathit{reached\text{-}e} \leftarrow 0$;
    while $\mathit{reached\text{-}e} < \mathit{total\text{-}d}$ do
        $v \leftarrow$ EXTRACT-MIN($Q$);
        SCAN($v$);
        if $e_f(v) > 0$ then
            $\mathit{reached\text{-}e} \leftarrow \mathit{reached\text{-}e} + e_f(v)$;
        $\mathit{last\text{-}key} \leftarrow \kappa[v]$;
    while $Q \neq \emptyset$ do
        $v \leftarrow$ EXTRACT-MIN($Q$);
        if $\kappa[v] = \mathit{last\text{-}key}$ then
            SCAN($v$);
        else
            $\kappa[v] \leftarrow \mathit{last\text{-}key} + 1$;
    for each $v$ in the network do
        $p(v) \leftarrow p(v) - \kappa[v] \times \epsilon$;
end.
```

```
procedure SCAN($v$);
    for each arc $(u, v) \in E_f$ into $v$ do
        if $\kappa[v] + \ell(u, v) < \kappa[u]$ then
            $\kappa[u] \leftarrow \kappa[v] + \ell(u, v)$;
            DECREASE-KEY($Q, u, \kappa[u]$);
end.
```

Figure 3.8: The Global Update Algorithm for Assignment

A global update operation begins by iteratively applying the node-scanning step from Dijkstra's algorithm to compute a key $\kappa[v]$ for each node v. Node scanning ends as soon as the algorithm has scanned all nodes v from which an excess can be reached in the (original) admissible graph; this criterion is enough to ensure ϵ-optimality and an admissible path to a deficit from each excess (in the new admissible graph) while bounding the range of possible key values. After the node scanning loop, the operation subtracts the quantity $\kappa[v] \times \epsilon$ from $p(v)$ for each node v.

3.6.2 Properties of the Global Update Algorithm

We need some basic results that will help us establish important properties of global updates.

Lemma 3.6.2 *Global updates preserve ϵ-optimality of the current pseudoflow and price function.*

Proof: Consider a residual arc (v, w). If neither v nor w is scanned the update reduces $p(v)$ and $p(w)$ by the same amount, and so the arc obeys ϵ-optimality after the update if it did before. If v is scanned and w isn't scanned the algorithm increases $c_p(v, w)$ because the prices of unscanned nodes decrease by more than the prices of scanned nodes. Finally, if w is scanned the algorithm ensures that $\kappa[v] \leq \kappa[w] + \ell(v, w)$, and so ϵ-optimality follows directly from the definition of ℓ. ∎

Lemma 3.6.3 *A global update cannot increase the price of any node.*

Proof: Immediate from the algorithm. ∎

Lemma 3.6.4 *Let $(v, w) \in E_f$ when a global update is performed. After the global update, (v, w) is admissible if and only if $\ell(v, w) = \kappa[v] - \kappa[w]$.*

Proof: Straightforward from the definitions of ℓ and κ. ∎

We will say a node v was *reached along an arc* (v, w) if $\kappa[v]$ is changed while w is being scanned.

Lemma 3.6.5 *If a node v is scanned, either $e_f(v) < 0$ or v was reached along some arc (v, w) when w was scanned, such that the arc obeys $\kappa[v] = \kappa[w] + \ell(v, w)$.*

Proof: This lemma would be trivial if not for the fact that the initial κ values given to all nodes outside D are finite. To prove the lemma, it will suffice to show that no node v with $\kappa[v] = 2\alpha n$ is ever scanned. This amounts to showing that *last-key* $< 2\alpha n$ when the condition *reached-e* $=$ *total-d* becomes true, ending the **while** loop. This fact is a straightforward consequence of Lemma 3.6.1 and the definition of ℓ. ∎

Lemma 3.6.6 *If a node v is scanned during a global update, then immediately after the update there exists an admissible path from v to some node in D.*

Proof: By induction on the number of scanning steps. The basis step is trivial, since the empty path from a deficit node to itself is trivially admissible. The induction follows because if a node v is scanned, it was reached along some arc (v, w) when w was scanned (by Lemma 3.6.5), such that the arc obeys $\kappa[v] = \kappa[w] + \ell(v, w)$. After the global update (v, w) will be admissible by Lemma 3.6.4, and there is an admissible path from w to a node of D by the induction hypothesis. ■

The following three theorems encapsulate properties of the global update algorithm that are required for later analysis.

Theorem 3.6.7 *The global update algorithm preserves acyclicity of the admissible graph G_A.*

Proof: Suppose that immediately after a global update, G_A contains a cycle. By Lemma 3.6.6, either the cycle lies completely outside the set of scanned nodes, or all the nodes on the cycle have shortest paths to deficits that include the entire cycle (note that no admissible arcs leave the set of scanned nodes). In either case, all arcs a on the cycle must have $\ell(a) = 0$. Therefore, all the cycle's arcs were admissible before the global update. ■

Theorem 3.6.8 *Let v be some node with $e_f(v) > 0$, and let w be some node reachable from v in G_A immediately after a global update. Then there exists a path in G_A from w to some node x with $e_f(x) < 0$.*

Proof: Immediate from Lemma 3.6.6 along with the fact that if a node is not scanned, it is not reachable from any excess after the update. ■

Theorem 3.6.9 *The global update algorithm runs in $O(m)$ time.*

Proof: Clearly the amount of time spent scanning a node is proportional to a constant plus the degree of the node, since each incident arc is examined once during a scan.

Because each node is scanned only once during an update, the running time is $O(n + m) = O(m)$ plus the amount of time spent selecting nodes to scan. The total time for node selection is $O(n)$ if, using the main idea from Dial's work [15], we implement our priority queue as an array of $2\alpha n$ doubly-linked lists. ∎

Finally, we observe that global updates preserve the basic analysis of *refine* because they do not alter the price of any node v with $e_f(v) < 0$ and because they do not increase node prices.

3.7 Weighted Bad Example: Minimum Change Discharge Algorithm without Global Updates

We present a family of assignment instances on which we show *refine* without global updates performs $\Omega(nm)$ work in the first scaling iteration, under the minimum change discharge selection rule. Hence this family of matching networks suffices to show that global updates account for an asymptotic difference in running time.

The family of assignment instances on which we show *refine* without global updates takes $\Omega(nm)$ time is structurally the same as the family of bad examples we used in the unweighted case, except that they are have two additional nodes and one additional edge. The costs of the edges present in the unweighted example are zero, and there are two extra nodes connected only to each other, sharing an edge with cost α. These two nodes and the edge between them are present only to establish the initial value of ϵ and the costs of arcs introduced in the reduction, and are ignored in our description of the execution.

At the beginning of the first scaling iteration, $\epsilon = \alpha$. The iteration starts by setting $\epsilon = 1$. From this point on the execution is similar to the execution of the Minimum Distance Discharge algorithm given in Section 3.3, but the details differ because of the asymmetric definitions of ϵ-optimality and admissibility we use in the weighted case.

```
1. Initialization establishes $|X|$ units of excess, one at each node of $X$.
2. While some node $w \in Y$ has no excess,
   2.1. a unit of excess moves from a node of $X$ to $w$;
   2.2. $w$ is relabeled so that $p(w) = -2$.
   Remark: Now every node of $Y$ has one unit of excess.
3. Active nodes in $X$ are relabeled one-by-one so that each has price $-2$.
4. A unit of excess moves from the most recently relabeled node of $X$ to a node of $Y$,
   then to $t$, and on to cancel a unit of deficit at $s$.
5. While more than one node of $Y$ has excess,
   5.1. A unit of excess moves to $t$ and thence to $s$ from a node of $Y$;
6. The remaining unit of excess at a node of $Y$ moves to a node $v \in X$ with $p(v) = 0$,
   and $v$ is relabeled so that $p(v) = -2$.
7. $\ell \leftarrow 1$; While $\ell \leq \alpha n/2 - 1$,
         Remark: All excesses are at nodes of $X$, and these nodes have price $-2\ell$; all
         other nodes in $X$ have price $-2\ell + 2$; all nodes in $Y$ have price $-2\ell$.
   7.1. A unit of excess is selected, and while some node $x \in X$ has $p(x) = -2\ell + 2$,
         • the selected unit moves from some active node $v$ to $w$, a neighbor of $x$ in
           $G_f$ (for a given $x$ there is a unique such $w$);
         • the unit of excess moves from $w$ to $x$;
         • $x$ is relabeled so $p(x) = -2\ell$.
         Remark: Now all nodes in $X \cup Y$ have price $-2\ell$; all excesses are at nodes of $X$.
   7.2. While some node $w \in Y$ has $p(w) = -2\ell$ and some node $v \in X$ has $e_f(v) = 1$,
         • a unit of excess moves from $v$ to $w$;
         • $w$ is relabeled so $p(w) = -2\ell - 2$.
         Remark: The following loop is executed only if $|X| < 2|Y|$. All active nodes in
         $Y$ have price $-2\ell - 2$, and all other nodes in $Y$ have price $-2\ell$.
   7.3. If a node in $Y$ has price $-2\ell$, a unit of excess is selected, and while some node
         $y \in Y$ has $p(y) = -2\ell$,
         • the selected unit moves from some $w \in Y$ with $e_f(w) = 1$ to $v \in X$ with
           $p(v) = -2\ell$, and then to $y$;
         • $y$ is relabeled so $p(y) = -2\ell - 2$.
         Remark: The following loop is executed only if $|X| > 2|Y|$.
   7.4. For each node $v \in X$ with $e_f(v) = 1$,
         • $v$ is relabeled so $p(v) = \max\{-2\ell - 2, -\alpha n\}$.
   7.5. For each node $w \in Y$ with $e_f(w) = 1$,
         • a unit of excess moves from $w$ to $v \in X$ with $p(v) = -2\ell$;
         • $v$ is relabeled so $p(v) = \max\{-2\ell - 2, -\alpha n\}$.
   7.6. $\ell \leftarrow \ell + 1$.
8. Excesses move one-by-one from active nodes in $X$ (which have price $-\alpha n$) to $s$.
```

Figure 3.9: The Minimum Change Discharge execution on bad examples.

Figure 3.9 details an execution of the Minimum Change Discharge algorithm without global updates. As in the unweighted case, every relabeling changes a node price by at most two, and the algorithm does $\Omega(n^2)$ relabelings. Consequently, the relabelings require $\Omega(nm)$ work, and we have the following theorem:

Theorem 3.7.1 *For any function $m(n)$ in the range $n \leq m(n) < n^2/4$, there exists an infinite family of instances of the assignment problem having $\Theta(n)$ nodes and $\Theta(m(n))$ edges on which the Minimum Change Discharge implementation of* refine *without global updates runs in $\Omega(nm(n))$ time.*

3.8 The First Scaling Iteration

Let G be the network produced by reducing an assignment problem instance to the minimum cost circulation problem as in Section 2.2.2. When *refine* initializes by saturating all negative arcs in this network, the only deficit created will be at s by our assumption that the input costs are nonnegative.

For a pseudoflow f in G, define $\mathcal{E}_t(f)$ to be the amount of f's excess that can reach s by passing through t. $\mathcal{E}_t(f)$ corresponds to the residual flow value in the unweighted case (see Section 3.1).

The $(\alpha\epsilon)$-optimality of the initial flow and price function played an important role in the proof of Lemma 3.5.5, specifically by lower-bounding the initial cost of any arc that currently carries a unit of flow. In contrast, the first scaling iteration may have many arcs that carry flow and have extremely negative costs relative to ϵ, specifically those arcs of the form (s, v) introduced by the reduction. But to counter this difficulty, the first iteration has an advantage that later iterations lack: an *upper* bound (in terms of ϵ) on the initial cost of *every* residual arc in the network. Specifically, recall that the value of ϵ in the first iteration is C/α, where C is the largest cost of an edge in the given assignment instance. So for any arc a other than the (v, s) arcs introduced by the reduction, $c(a) \leq \alpha\epsilon$ in the first scaling iteration.

Lemma 3.8.1 *At any point during the first execution of* refine*, $\mathcal{E}_t \times \Gamma_{\max} \leq n(2+\alpha)$.*

Proof: Let f, p be the pseudoflow and price function at the most recent point when $\Gamma(f,p) = \Gamma_{\max}$. Let f^* be a minimum cost circulation in G, and let $f' = f^* - f$. Recall that the costs on the (s,v) arcs are negative enough that f^* must correspond to a matching of maximum cardinality. Therefore, f' moves $\mathcal{E}_t(f)$ units of f's excess to s through t, and returns the remainder to s without its passing through t. Now $-f'$ is a pseudoflow in G_{f^*}, and can be decomposed into cycles and paths according to Lemma 3.5.3; as in the proof of Lemma 3.1.4, let $\mathcal{P}$ denote the induced decomposition of f'. Let $\mathcal{Q} \subseteq \mathcal{P}$ be the set of paths that pass through t, and note that $\mathcal{E}_t(f) = |\mathcal{Q}|$. Let $e_f^t(v)$ denote the number of paths of $\mathcal{Q}$ beginning at node v. The only deficit in f is at s, so $e_f^t(v)$ is precisely the amount of v's excess that reaches s by passing through t if we imagine augmenting f along the paths of $\mathcal{P}$. Of particular importance is that no path in $\mathcal{Q}$ uses an arc of the form (s,v) or (v,s) for $v \neq t$.

Observe that

$$\mathcal{E}_t(f) \times \Gamma_{\max} \leq \sum_{e_f^t(v)>0} e_f^t(v)\delta(v),$$

so by the definition of δ,

$$\epsilon \times \mathcal{E}_t(f) \times \Gamma_{\max} \leq -\sum_{e_f^t(v)>0} e_f^t(v)p(v).$$

Now note that for any path P from v to s, we have $p(v) = c_p(P) - c(P)$ because $p(s) = 0$. Every arc used in the decomposition $\mathcal{P}$ appears in G_f. By ϵ-optimality of f and Lemma 3.5.4, each of the n or fewer arcs a in G_f with negative reduced cost has $c_p(a) \geq -2\epsilon$, so we have $\sum_{P\in\mathcal{Q}} c_p(P) \geq -2n\epsilon$. Next, we use the upper bound on the initial costs to note that $\sum_{P\in\mathcal{Q}} c(P) \leq \alpha n\epsilon$, so

$$\epsilon \times \mathcal{E}_t(f) \times \Gamma_{\max} \leq -\sum_{e_f^t(v)>0} e_f^t(v)p(v) \leq 2n\epsilon + \alpha n\epsilon = n(2+\alpha)\epsilon,$$

and the lemma follows. ∎

Lemma 3.8.2 *At any point during the first execution of* refine, $\mathcal{E} \times (\Gamma_{\max} - \alpha n) \leq n(2+\alpha)$.

Proof: Essentially the same as the proof of Lemma 3.8.1, except that if $\Gamma_{\max} > \alpha n$, each path from an excess to the deficit at s will include one arc of the form (v, s), and each such arc has original cost $-nC = -\alpha n\epsilon$. ∎

Lemmas 3.8.1 and 3.8.2 allow us to split the analysis of the first scaling iteration into four stages, much as we did with the Minimum Distance Discharge algorithm for matching. Specifically, the analysis of Section 3.5 holds up until the point where $\Gamma_{\max} \geq \alpha n$, with Lemma 3.8.1 taking the place of Lemma 3.5.5. Straightforward extensions of the relevant lemmas show that the algorithm does $O(km + n^2/k)$ work when $\Gamma_{\max} \in [\alpha n, \alpha n + k]$, and when $\Gamma_{\max} > \alpha n + k$, Lemma 3.8.2 bounds the algorithm's work by $O(nm/k)$. The balancing works as before: choosing $k = \Theta(\sqrt{n})$ gives a bound of $O(\sqrt{n}m)$ time for the first scaling iteration.

3.9 Weighted Global Updates in Practice: Effects on a Generic Implementation

In this section, we investigate the practical effects of global updates on the performance of a push-relabel implementation that solves the minimum cost flow problem. The heuristic's benefits are known for many classes of minimum cost flow problems [25], but it is interesting to verify that the technique helps even when the domain of discussion is restricted to assignment problems.

The implementation we study is a modified form of a cost-scaling code developed in [25]; in the code with global updates switched on, an update was performed after every $1 + \lfloor 15.4n \rfloor$ relabelings. The minimum cost flow code uses a slightly different reduction from assignment to minimum cost flow from the one outlined in Chapter 2. We defer a discussion of the different reduction and the reasons for it to Chapter 4, where we present a detailed performance study of a set of implementations. We also delay until Chapter 4 details of the various problem classes used for our tests, since the aim here is simply to show quickly that global updates provide a substantial speed advantage over an implementation that uses no heuristics. For the same reason, we limit the number of problem classes discussed here; performance gains are comparable on the remaining classes described in Chapter 4.

3.9.1 Running Times and Discussion

The platform on which we conducted these tests is the same as we used for the more detailed study in Chapter 4. See Section 4.4 for a description of the system and measurement techniques. For consistency with the presentation there, we identify problem sizes according to the number of nodes in $|X|$, *i.e.*, on one side of the graph. The total number of nodes in each instance is twice this figure.

The times and sample deviations reported are in seconds, and are computed over a sample of three instances for each class and problem size.

Low-Cost Problems

	No GU		With GU		
$\lvert X\rvert$	time	s	time	s	speedup
1024	26.5	13	4.1	0.2	6.5
2048	77.5	14	11.5	0.6	6.7
4096	205	55	27.9	1.0	7.4
8192	565	105	66.4	3.8	8.5
16384	2196	511	188	19	11.7
32768	5839	2250	501	6.9	11.7

Figure 3.10: Running time comparison on the Low-Cost class

On low-cost problems, the code using global updates has a substantial speed advantage that grows with problem size. Also, the code without global updates is less robust in the sense that the running times show high variances.

Fixed-Cost Problems

As in the case of low-cost problems, the use of global updates makes running times more predictable as well as smaller on fixed-cost problems. The speed advantage of the implementation using global updates appears to remain roughly constant on this class as problem size increases.

	No GU		With GU		
$\lvert X\rvert$	time	s	time	s	speedup
128	2.4	1.4	0.58	0.03	4.1
256	4.6	1.6	2.4	0.11	2.0
512	41	18	11	0.48	3.7
1024	122	53	57	3.1	2.2
2048	917	626	278	14.2	3.3

Figure 3.11: Running time comparison on the Fixed-Cost class

Geometric Problems

	No GU		With GU		
$\lvert X\rvert$	time	s	time	s	speedup
128	5.6	3.0	3.1	0.06	1.8
256	24.5	15	14.5	1.1	1.7
512	342	380	76.3	5.1	4.5
1024	846	229	361	19	2.3

Figure 3.12: Running time comparison on the Geometric class

Global updates appear to give a small constant factor speedup on geometric problems, and also greatly reduce the variance in running times.

Dense Problems

	No GU		With GU		
$\lvert X\rvert$	time	s	time	s	speedup
128	5.9	2.6	1.9	0.2	3.2
256	48	24	9.0	1.0	5.3
512	278	146	45	3.5	6.2
1024	2097	1160	232	9.7	9.0

Figure 3.13: Running time comparison on the Dense class

The speed advantage of the code using global updates grows with problem size on dense instances. Once again, global updates show greater robustness by keeping the running time variance small.

As in the unweighted case, we observe a substantial speed improvement owing to the use of global updates in practice, compared against the same implementation without them. On every problem class we tested, global updates provide an expected increase in performance over a generic implementation that does not use them.

3.10 Chapter Summary

We have given algorithms that achieve the best time bounds known for bipartite matching, namely $O\left(\sqrt{n}m\frac{\log(n^2/m)}{\log n}\right)$, and for the assignment problem in the cost-scaling context, namely $O\left(\sqrt{n}m\log(nC)\right)$. We have also given examples to show that without global updates, the algorithms perform worse. Hence we conclude that global updates can be a useful tool in theoretical development of algorithms.

In conjunction with the theoretical results, we briefly gave running time data to show that global updates are of practical value in the context where our analysis pertains. This is no surprise given the success of global updates in earlier implementation studies for maximum flow and minimum cost flow, but it is nevertheless interesting to check because earlier studies using global updates have not focused on bipartite problems.

We have shown a family of assignment instances on which *refine* without global updates performs poorly, but the poor performance seems to hinge on details of the reduction so it happens only in the first scaling iteration. An interesting open question is the existence of a family of instances of the assignment problem on which *refine* uses $\Omega(nm)$ time in *every* scaling iteration.

Finally, it is natural to ask whether global updates give an improved worst-case time bound in the case of general (as opposed to bipartite) networks with general capacities. This question remains open.

Chapter 4

Implementation Study: Efficient Cost-Scaling Code for Assignment

In this chapter we study implementations of the scaling push-relabel method in the context of the assignment problem. We use ideas from minimum-cost flow codes [7, 25, 29], ideas from assignment codes [5, 8, 6], and ideas of theoretical work on the push-relabel method for the assignment problem [30], as well as new techniques aimed at improving practical performance of the method. We develop several CSA (**C**ost **S**caling **A**ssignment) codes based on different heuristics which improve the code performance on many problem classes. The "basic" code CSA-B uses only the most straightforward heuristics, the CSA-Q code uses a "quick-minima" heuristic in addition, and the CSA-S code uses a "speculative arc fixing" heuristic. Another outcome of our research is a better understanding of cost-scaling algorithm implementations, which may lead in turn to improved cost-scaling codes for the minimum-cost flow problem.

We compare the performance of the CSA codes to four other codes that represent the previous state of the art in implementations for the assignment problem: SFR10, an implementation of the auction method for the assignment problem [8]; SJV and DJV, implementations of Jonker and Volgenant's shortest augmenting path method

[39] tuned for sparse and dense graphs respectively; and ADP/A, an implementation of the interior-point method specialized for the assignment problem [48]. We make the comparison over classes of problems from generators developed for the First DIMACS Implementation Challenge [38][1] and on problems obtained from digital images as suggested by Don Knuth [42]. Overall, the best of our codes is CSA-Q. It outperforms ADP/A on all problem instances in our tests, outperforms SFR10 on all except one class, and outperforms SJV and DJV on large instances in every class. Although our second-best code, CSA-S, is somewhat slower than CSA-Q on many problem classes, it is usually not much slower and it outperforms CSA-Q on three problem classes, always outperforms ADP/A, is worse than SFR10 by only a slight margin on one problem class and by a noticeable margin on only one problem class, and loses to the Jonker-Volgenant codes only on one class and on small instances from two other classes. While we use the CSA-B code primarily to gauge the effect of heuristics on performance, it is worth noting that it outperforms ADP/A in all our tests, the Jonker-Volgenant codes on large instances from all but one class, and SFR10 on all but one class of problems we tested.

ADP/A	no better than $O(mn^{1.5}\log(nC))$
JV codes	$O(n^3)$
SFR10	$O(n^2 m\log(nC))$
CSA codes	$O(nm\log(nC))$

Figure 4.1: Worst-case bounds for the assignment codes

Figure 4.1 shows the best bounds we know on the worst-case running time of each of the implementations in our study. Only the Jonker-Volgenant codes compete with the CSA implementations on the basis of worst-case running time; all the other time bounds are asymptotically worse.

We begin our exposition in Section 4.1 by discussing the basic ideas behind our implementations, and move on to a discussion of the heuristics we evaluated during

[1]The DIMACS benchmark codes, problem generators, and other information we refer to are available by anonymous `ftp` from `dimacs.rutgers.edu`

the development of the CSA codes in Section 4.2. In subsequent sections, we outline the experiments we ran to evaluate our codes' performance, and we give the results of the experiments and our conclusions.

4.1 Implementation Fundamentals

In this section, we discuss some general considerations that must be taken into account in any push-relabel implementation for the assignment problem. Such a discussion forms the background for the coming discussion of implementation heuristics and their role in the practical performance of algorithms.

All the push-relabel implementations we study use some version of the "forward star" data structure (see [1]) to represent the input graph. The information pertaining to each node in the graph is collected together in an array entry indexed by the identifier for the node, and similarly for the arcs. Arcs directed out of a particular node are stored contiguously. Such a structure has obvious benefits with respect to locality of reference.

General minimum cost circulation implementations [25] typically store each arc explicitly, and enforce flow antisymmetry on symmetric arcs by linking paired arcs together. Many implementations to solve the assignment problem, even among those that reduce the problem to minimum cost circulation (or minimum cost flow), use a more compact representation. The three implementations we develop and study in this chapter store only "forward" arcs, although some of the heuristics we evaluated (*e.g.*, global updates) required reverse arcs as well for efficiency.

We develop our implementations with the view that they solve a class of minimum cost flow problems that arise through a reduction from the assignment problem. We do not hesitate to take advantage of the special characteristics of assignment, but we find the principles and the details easier to understand and express from the flow-oriented point of view. The ideas can be viably interpreted from a matching-oriented perspective, but we prefer the more general viewpoint since it supports a unified understanding of related problems.

For our implementations we use a reduction to minimum cost flow that differs from the one detailed in Chapter 2. This decision is driven by several considerations: The theoretical context of our global update study called for as general a reduction as possible, one that assumed nothing about the structure of the given assignment problem and that made it straightforward to demonstrate the separation in the algorithms' performance with and without the updates. In an implementation, our needs are somewhat different since we mean to design appealing code to solve assignment problems and to compare our code with other implementations from the literature on the basis of performance. To make a performance comparison as meaningful as we can, we wish to use the same conventions as our competitors with regard to the given assignment instances; this uniformity will help ensure that background assumptions are not responsible for observed differences in speed.

For the purposes of our codes, the reduction we use is to the *transportation problem*, also known as minimum cost flow with supplies and demands. Given an assignment instance, we introduce a unit-capacity arc for each edge as before, and retain the given problem's costs. At each node of X we place a single unit of supply, and at each node of Y, we place a unit of demand. We seek a flow of minimum cost that satisfies these supplies and demands; such a flow corresponds in the obvious way to a perfect matching of minimum cost. Note that the transportation problem we get from this reduction is feasible if and only if the given assignment instance admits a perfect matching. We assume the existence of a perfect matching; the same assumption is made by the competing codes in our study.

It would be a straightforward matter to adapt our codes to a more general reduction such as the one in Chapter 2; this adaptation would dispense with the assumption of a perfect matching, and we believe the change would affect the code's performance only slightly.

4.2 Push-Relabel Heuristics for Assignment

The literature suggests a variety of techniques that make various contributions to improving the running time of push-relabel network flow implementations; many of these heuristics specialize to the assignment problem in meaningful ways, and we explore some of them here. Additionally, the special structure of the assignment problem suggests a few heuristics that do not owe their origins to those developed in more general contexts.

We define each heuristic that we implemented and explore the issues surrounding its implementation in the context of the assignment problem. We do not use all the heuristics in our final CSA implementations because not all of them improve the assignment codes' performance.

4.2.1 The Double-Push Operation

We describe a heuristic that determines how the *push* and *relabel* operations combine at a low level. The *double-push* operation is roughly similar to a sequential version of the match-and-push procedure from [30], and the operation is designed to improve the implementation's performance in two ways. First, it allows the use of simplified data structures and can be implemented efficiently. Second, it relabels nodes "aggressively" in some sense. We discuss these two advantages in more detail below, after we define the operation and prove its basic properties.

The *double-push* operation applies to an active node $v \in X$. The operation assumes that at the beginning of a double-push, all nodes $w \in Y$ with $e_f(w) \geq 0$ have an admissible outgoing arc, and our algorithms that use *double-push* will maintain this property as an invariant. The double-push operation begins by processing v: it relabels v, pushes flow from v along an admissible arc (v, w), and then relabels v again. If $e_f(w)$ is positive after the push from v, the operation pushes a unit of flow along an admissible arc (w, x). Finally, double-push relabels w.

Analysis of Double-Push

The following lemma establishes the correctness of the double-push operation.

Lemma 4.2.1 *The* double-push *operation maintains the invariant that every node $w \in Y$ with $e_f(w) \geq 0$ has an outgoing admissible arc.*

Proof: Consider a double-push from v into w. The last action performed by this double-push is to relabel w, and therefore immediately after the double-push, there is an admissible arc leaving w. Now let x be some node (x can be v) such that $(w, x) \in E_f$. Nodes in X never have deficits by the asymmetric definition of ϵ-optimality, so until another double-push into w occurs, (w, x) is the only residual arc into x. Therefore, x cannot be active before another double-push into w, and hence x is not relabeled until such a double-push. This fact immediately implies that the arc (w, x) remains admissible until the next double-push into w. ■

The standard analysis of push-relabel algorithms assumes that the *relabel* operation is applied only to active nodes. Because double-push relabels nodes that are not active (*i.e.*, the second relabeling of v in the above description), we give a straightforward reanalysis of the push-relabel algorithm to show that double-push leaves the theoretical performance bounds intact. The analysis will use the following obvious lemma.

Lemma 4.2.2 *The* double-push *operation maintains the invariant that every node $w \in Y$ has $e_f(w) \in \{-1, 0\}$, except while the operation is in progress. Equivalently,* double-push *maintains the invariant that every node $w \in Y$ has at most one outgoing residual arc, except while the operation is in progress.*

For each $w \in Y$ with $e_f(w) = 0$ we define $\mu(w)$ to be the unique node such that $f(\mu(w), w) = 1$. If $e_f(w) = 0$ for $w \in Y$, we say w is *matched* to $\mu(w)$. If $e_f(w) = -1$, we say w is *unmatched*.

Lemma 4.2.3 *A double-push operation decreases the price of a node $w \in Y$ by at least 2ϵ.*

Proof: Just before the double-push, w is either unmatched or matched.

In the first case, the flow is pushed into w and at this point the only residual arc out of w is the arc (w, v). Just before that the double-push relabeled v and $c_p(v, w) = 0$. Next double-push relabels w and $p(w)$ decreases by 2ϵ.

In the second case, the flow is pushed to w and at this point w has two outgoing residual arcs, (w, v) and $(w, \mu(w))$. As we have seen, $c_p(v, w) = 0$ before v's second relabeling, and $c_p(\mu(w), w) = 2\epsilon$. After the second relabeling of v, double-push pushes flow from w to $\mu(w)$ and relabels w, reducing $p(w)$ by at least 2ϵ. ■

```
DOUBLE-PUSH($v$).
    let $(v,w)$ and $(v,z)$ be $v$'s outgoing residual arcs with the smallest and
        the second-smallest reduced costs;
    PUSH($v,w$);
    $p(v) = -c'_p(v,z)$;
    if $e_f(w) > 0$
        PUSH($w,\mu(w)$);
    $\mu(w) = v$;
    $p(w) = p(v) + c(v,w) - 2\epsilon$;
end.
```

Figure 4.2: Efficient implementation of double-push

The following corollary is immediate from Lemma 4.2.3 and the fact that the price of an active node can change by at most $O(n\epsilon)$ during an execution of *refine*.

Corollary 4.2.4 *There are $O(n^2)$ double-pushes per refine, and they require $O(nm)$ time.*

Efficient Implementation of Double-Push

Lemma 4.2.2 benefits not only the analysis; it allows an important simplification of the implementation's underlying data structure as well. In particular, it eliminates the need to store explicitly the value of each node's flow excess and limits the residual outdegree of nodes in Y to be at most one. We will reap the benefit that relabeling an active node on the right can be very fast, since it requires examining only one residual arc. Rather than store excesses and deficits explicitly, it will suffice to have some way of determining the number (zero or one) of residual arcs into (for nodes in X) or out of (for nodes in Y) each node. Specifically, for each node $w \in Y$ with $e_f(w) = 0$, our implementations will maintain a pointer to $\mu(w)$; for nodes with deficit, this pointer will take a special value to indicate that the corresponding node is unmatched.

Suppose we apply double-push to a node v. Let (v,w) and (v,z) be the arcs out of v with the smallest and the second-smallest reduced costs, respectively. These arcs can by found by scanning the adjacency list of v once. The effects of double-push on v are equivalent to pushing flow along (v,w) and setting $p(v) = -c'_p(v,z)$. To relabel w, we set $p(w) = p(v) + c(v,w) - 2\epsilon$. This implementation of double-push is summarized in Figure 4.2.

It is not necessary to maintain the prices of nodes in X explicitly; for $v \in X$, we can define $p(v)$ implicitly by $p(v) = -\min_{(v,w)\in E}\{c'_p(v,w)\}$ if $e_f(v) = 1$ and $p(v) = c'(v,w) + 2\epsilon$ if $e_f(v) = 0$ and (v,w) is the unique arc with $f(v,w) = 1$. One can easily verify that using implicit prices is equivalent to using explicit prices in the above implementation. The only time we need to know the value of $p(v)$ is when we relabel w in double-push, and at that time $p(v) = -c'_p(v,z)$ which we compute during the previous relabel of v. Maintaining the prices implicitly saves memory and time. The implementation of the double-push operation with implicit prices is similar to the basic step of the auction algorithm of [5].

The main advantage of double-push seems to be that it allows simplification of the main data structures because units of excess never collide, and that it maintains an "aggressive" price function by using every node scan to perform, in effect, three relabelings. In our preliminary tests, double-push showed such a decisive advantage over other low-level ways of combining *push* and *relabel* operations that all the CSA implementations use it.

4.2.2 The kth-best Heuristic

Some implementation studies [5, 8] have suggested that in the case of the assignment problem, the *relabel* operation can often be sped up by exploiting the fact that arcs with low reduced cost are likely to be admissible in the future.

The kth-best heuristic is aimed at reducing the number of scans of arc lists of nodes in X, and is a generalization of that of the 3rd-best scheme outlined in [5]. Recall that we scan the list of v to find the arcs (v,w) and (v,z) with the smallest and second-smallest values of the partial reduced cost. Let $k \geq 3$ be an integer. When we scan the list of $v \in X$, we compute the kth-smallest value K of the partial reduced costs of the outgoing arcs and store the $k-1$ arcs with the $k-1$ smallest partial reduced costs. The node prices monotonically decrease during refine, hence during the subsequent double-push operations we can first look for the smallest and the second-smallest arcs among the stored arcs whose current partial reduced cost is at most K. We need to scan the list of v again only when all except possibly one of the saved arcs have partial reduced costs greater than K.

The kth-best heuristic was usually beneficial in our tests; the best value for k seems to be 4.

4.2.3 Arc Fixing

Because the amount by which a node price can change is bounded in each iteration and this bound on the per-iteration price change per node decreases geometrically (along with the scaling parameter ϵ), the algorithm can reach a point at which some arcs have such large reduced costs that they can never be admissible for the remainder of the algorithm. The algorithm will never alter the flow on such an arc, so the arc is said to be *fixed*. The following theorem is from [32] (see also [50]).

Theorem 4.2.5 *Let $\epsilon > 0$, $\epsilon' \geq 0$ be constants. Suppose that a circulation f is ϵ-optimal with respect to a price function p, and that there is an arc (v, w) such that $|c_p(v, w)| \geq n(\epsilon + \epsilon')$. Then for any ϵ'-optimal circulation f', we have $f(v, w) = f'(v, w)$.*

If the flow on a large number of arcs is fixed according to Theorem 4.2.5, the speed of an implementation can benefit from ignoring those arcs. We will say that such an implementation uses the *guaranteed arc fixing* heuristic. Recent implementations to solve minimum cost flow [21, 25] commonly presume that Theorem 4.2.5 gives quite a conservative guarantee in practice, and that arcs with moderately large positive reduced costs are unlikely to become admissible in the future.

The idea of the *speculative arc fixing* heuristic [21, 25] is to move arcs with reduced costs whose magnitudes exceed some threshold θ to a special list. These arcs are not examined by the double-push procedure but are examined as follows at a (relatively large) periodic interval. When the arc (v, w) is examined, if the ϵ-optimality condition is violated on (v, w), the algorithm modifies $f(v, w)$ to restore ϵ-optimality and moves (v, w) back to the adjacency list of v; if ϵ-optimality holds for (v, w) but $|c_p(v, w)|$ no longer exceeds θ, the arc (v, w) is simply moved back to the adjacency list. Recovery from the situation in which an arc violates ϵ-optimality tends to be very expensive because large changes to the price function are often required.

Both guaranteed and speculative arc fixing tended to improve running times on the classes of problems we tested, speculative arc fixing by a significantly larger margin when θ was chosen carefully. A relatively common scenario in our tests seemed to be that for some $v \in X$ with $e_f(v) = 0$, all the residual arcs outgoing from v would be presumed fixed (or known fixed, in the case of guaranteed fixing). Since any feasible flow in the network must induce a matching, we can consider fixing all arcs incident to w in this situation as well. In the case of guaranteed arc fixing, this action proved not to be advantageous because of the time spent maintaining the additional information required to find the arcs incident to w. In the case of speculative arc fixing, this action greatly increased the risk that some arc would eventually violate ϵ-optimality. In both cases, the costs of fixing the additional arcs incident to w were too great.

4.2.4 Global Price Updates

Global updates have been studied extensively from a theoretical perspective in Chapter 3, and have been shown to improve the practical performance of codes for minimum cost flow, both when applied to a battery of minimum cost flow problems [25] and when applied to a suite of assignment problems (see Section 3.9).

In our CSA implementations, the code to perform global updates can take advantage of the simplified data structures and special properties of the residual graph that result when the algorithm uses the double-push operation with implicit prices as detailed in Section 4.2.1. Consequently the version of global updates we implemented for the assignment problem runs considerably faster than global update implementations for minimum cost flow. Even so, the double-push operation seems to maintain a sufficiently aggressive price function on its own; global price updates cannot reduce the number of *push* and *relabel* operations enough to improve the running time of the CSA implementations on the problem classes we tested. The CSA codes therefore do not use global updates.

4.2.5 Price Refinement

Price refinement determines at each iteration whether the current primal solution is actually ϵ'-optimal for some $\epsilon' < \epsilon$, and hence avoids unnecessary executions of *refine*. This idea has proven beneficial in implementations that solve the minimum cost flow problem in its full generality.

Price refinement can be implemented using a modified shortest-paths computation with primitives similar to those used to implement global updates, and like global updates, its code can be written to take advantage of the invariants of Lemmas 4.2.1 and 4.2.2. Even taking these properties into account, a typical price refinement iteration used more time in our tests than simply executing *refine* with *double-push*. Consequently, we do not use price refinement in the CSA implementations.

4.3 The CSA Codes

The efficiency of a scaling implementation depends on the choice of scale factor α. Although an earlier study [8] suggests that the performance of scaling codes for the assignment problem may be quite sensitive to the choice of scale factor, our observations are to the contrary. Spot checks on instances from several problem classes indicated that the running times seem to vary by a factor of no more than 2 for values of α between 4 and 40. We chose $\alpha = 10$ for our tests; different values of α would yield running times that are somewhat worse on some problem classes and somewhat better on others, but the difference is not drastic. We believe the lack of robustness alluded to in [8] may be due to a characteristic of the implementation of SFR10 and related codes. In particular, SFR10 contains an "optimization" that seems to terminate early scaling phases prematurely. Our codes run every scaling phase to completion as suggested by the theory.

The efficiency of an implementation of *refine* depends on the number of operations performed by the method and on the implementation details. All the CSA codes use the double-push operation with implicit prices and maintain the invariants of Lemmas 4.2.1 and 4.2.2.

The performance of the implementation depends on the strategy for selecting the next active node to process. We experimented with several operation orderings, including the minimum-distance strategy suggested in Chapter 3 and those suggested in [32]. Our implementation uses the LIFO ordering, *i.e.*, the set of active nodes is maintained as a stack. This ordering worked best in our tests; the FIFO and minimum distance orderings usually worked somewhat worse, although the difference was never drastic.

Our code CSA-B implements the scaling push-relabel algorithm using stack ordering of active nodes and the implementation of double-push with implicit prices mentioned above. Our code CSA-Q is a variation of CSA-B that uses the 4th-best heuristic. Our code CSA-S is a variation of CSA-B that uses the speculative arc fixing heuristic with the threshold $\theta = 4\epsilon n^{3/4}$. This threshold value was chosen empirically by spot-testing on a variety of problem instances.

4.4 Experimental Setup

All the test runs were executed on a Sun SparcStation 2 with a clock rate of 40 MHz, 96 Megabytes of main memory, and a 64-Kilobyte off-chip cache. We compiled the SFR10 code supplied by David Castañon with the Sun Fortran-77 compiler, release 2.0.1 using the `-O4` optimization switch[2]. We compiled the DJV and SJV codes supplied by Jianxiu Hao with the Sun C compiler release 1.0, using the `-O2` optimization option. We compiled our CSA codes with the Sun C compiler release 1.0, using the `-fast` optimization option; each choice seemed to yield the fastest execution times for the code where we used it. Times reported here are Unix *user* CPU times, and were measured using the `times()` library function. During each run, the programs collect time usage information after reading the input problem and initializing all data structures and again after computing the optimum assignment; we take the difference between the two figures to indicate the CPU time actually spent solving

[2]Castañon [8] recommends setting the initial "bidding increment" in SFR10 to a special value for problems of high density; we found this advice appropriate for the dense problem class, but discovered that it hurt performance on the geometric class. We followed Castañon's recommendation only on the class where it seemed to improve SFR10's performance.

C benchmarks *user* times	
Test 1	Test 2
2.7 sec	24.0 sec

FORTRAN benchmarks *user* times	
Test 1	Test 2
1.2 sec	2.2 sec

Figure 4.3: DIMACS benchmark times

the assignment problem.

To give a baseline for comparison of our machine's speed to others, we ran the DIMACS benchmarks `wmatch` (to benchmark C performance) and `netflo` (to benchmark FORTRAN performance) on our machines, with the timing results given in Figure 4.3. It is interesting (though neither surprising nor critical to our conclusions) to note that the DIMACS benchmarks do not precisely reflect the mix of operations in the codes we developed. Of two C compilers available on our system, the one that consistently ran our code faster by a few percent also ran the benchmarks more slowly by a few percent (the C benchmark times in Figure 4.3 are for code generated by the same compiler we used for our experiments). But even though they should not be taken as the basis for very precise comparison, the benchmarks provide a useful way to estimate relative speeds of different machines on the sort of operations typically performed by combinatorial optimization codes.

We did not run the ADP/A code, but because the benchmark times reported in [48] differ only slightly from the times we obtained on our machine, we conclude that the running times reported for ADP/A in [48] form a reasonable basis for comparison with our codes. Therefore, we report running times directly from [48]. As the reader will see, even if this benchmark comparison introduces a significant amount of error, our conclusions about the codes' relative performance are justified by the large differences in speed between ADP/A and the other codes we tested.

The DJV code is designed for dense problems and uses an adjacency-matrix data structure. The memory requirements for this code would be prohibitive on sparse problems with many nodes. For this reason, we included it only in experiments on problem classes that are dense. On these problems, DJV is faster than SJV by a factor of about 1.5. It is likely that our codes and the SFR10 code would enjoy a similar improvement in performance if they were modified to use the adjacency-matrix data structure.

We collected performance data on a variety of problem classes, many of which we took from the First DIMACS Implementation Challenge. Following is a brief description of each class; details of the generator inputs that produced each set of instances are included in Appendix A.

4.4.1 The High-Cost Class

Each $v \in X$ is connected by an edge to $2 \log_2 |V|$ randomly-selected nodes of Y, with integer edge costs uniformly distributed in the interval $[0, 10^8]$.

4.4.2 The Low-Cost Class

Each $v \in X$ is connected by an edge to $2 \log_2 |V|$ randomly-selected nodes of Y, with integer edge costs uniformly distributed in the interval $[0, 100]$.

4.4.3 The Two-Cost Class

Each $v \in X$ is connected by an edge to $2 \log_2 |V|$ randomly-selected nodes of Y, each edge having cost 100 with probability 1/2, or cost 10^8 with probability 1/2.

4.4.4 The Fixed-Cost Class

For problems in this class, we view X as a copy of the set $\{1, 2, \ldots, |V|/2\}$, and Y as a copy of $\{\, |V|/2+1, |V|/2+2, \ldots, |V|\,\}$. Each $v \in X$ is connected by an edge to $|V|/16$ randomly-selected nodes of Y, with edge (x, y), if present, having cost $100 \cdot x \cdot y$.

4.4.5 The Geometric Class

Geometric problems are generated by placing a collection of integer-coordinate points uniformly at random in the square $[0, 10^6] \times [0, 10^6]$, coloring half the points blue and the other half red, and introducing an edge between every red point r and every blue point b with cost equal to the floor of the distance between r and b.

4.4.6 The Dense Class

Like instances of the geometric class, dense problems are complete, but edge costs are distributed uniformly at random in the range $[0, 10^7]$.

4.4.7 Picture Problems

Picture problems, suggested by Don Knuth [42], are generated from photographs scanned at various resolutions, with 256 greyscale values. The set V is the set of pixels; the pixel at row r, column c is a member of X if $r + c$ is odd, and is a member of Y otherwise. Each pixel has edges to its vertical and horizontal neighbors in the image, and the cost of each edge is the absolute value of the greyscale difference between its two endpoints. Note that picture problems are extremely sparse, with an average degree always below four. Although picture problems are an abstract construct with no practical motivation, the solution to a picture problem can be viewed as a tiling of the picture with dominos, where we would like each domino to cover greyscale values that are as different as possible.

For our problems, we used two scanned photographs, one of the author, and one of Andrew Goldberg, the author's principal advisor.

4.5 Experimental Observations and Discussion

In the following tables and graphs we present performance data for the codes. Note that problem instances are characterized by the number of nodes on a single side, *i.e.*, *half* the number of nodes in the graph.

We report times on the test runs we conducted, along with performance data for the ADP/A code taken from [48]. The instances on which ADP/A was timed in [48] are identical to those we used in our tests. We give mean running times computed over three instances for each problem size in each class; in the two-cost and geometric classes we also give mean running times computed over 15 instances and sample deviations for each sample size. We computed sample deviations for each problem class and size, and observed that in most cases they were less than ten percent of the mean (often much less). The two exceptions were the two-cost and geometric classes, where we observed larger sample deviations in the running times for some of the codes. For these two classes we also collected data on 15 instances for each problem size. The sample statistics taken over 15 instances seem to validate those we observed for three instances. All statistics are reported in seconds.

4.5.1 The High-Cost Class

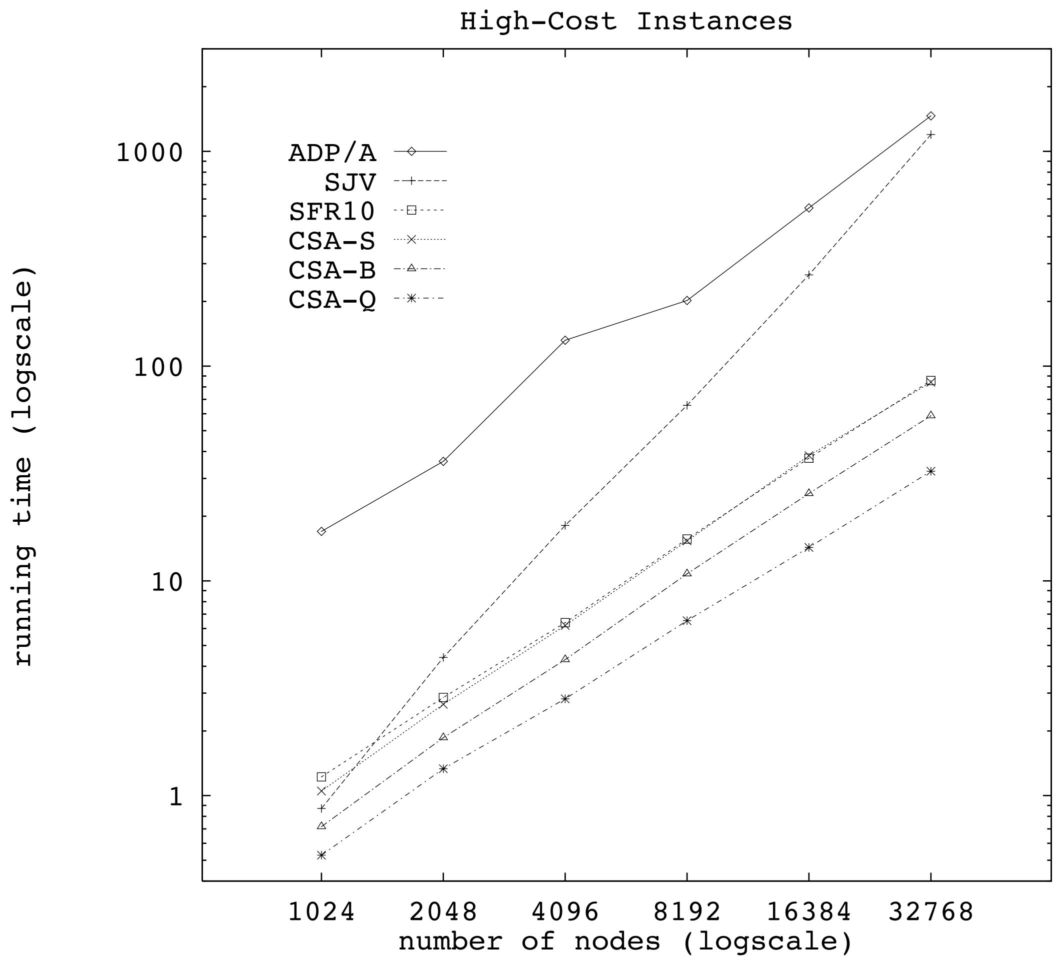
Figure 4.4 summarizes the timings on DIMACS high-cost instances. The kth-best heuristic yields a clear advantage in running time on these instances. CSA-Q beats CSA-B, its nearest competitor, by a factor of nearly 2 on large instances, and CSA-Q seems to have an asymptotic advantage over the other codes as well. The overhead of speculative arc fixing is too great on high-cost instances; the running times of CSA-S for large graphs are essentially the same as those of SFR10. SJV has the worst asymptotic behavior.

4.5.2 The Low-Cost Class

The situation here is very similar to the high-cost case: CSA-Q enjoys a slight asymptotic advantage as well as a clear constant-factor advantage over the competing codes. SJV has worse asymptotic behavior than the other codes on the low-cost class, just as it does on high-cost instances. See Figure 4.5.

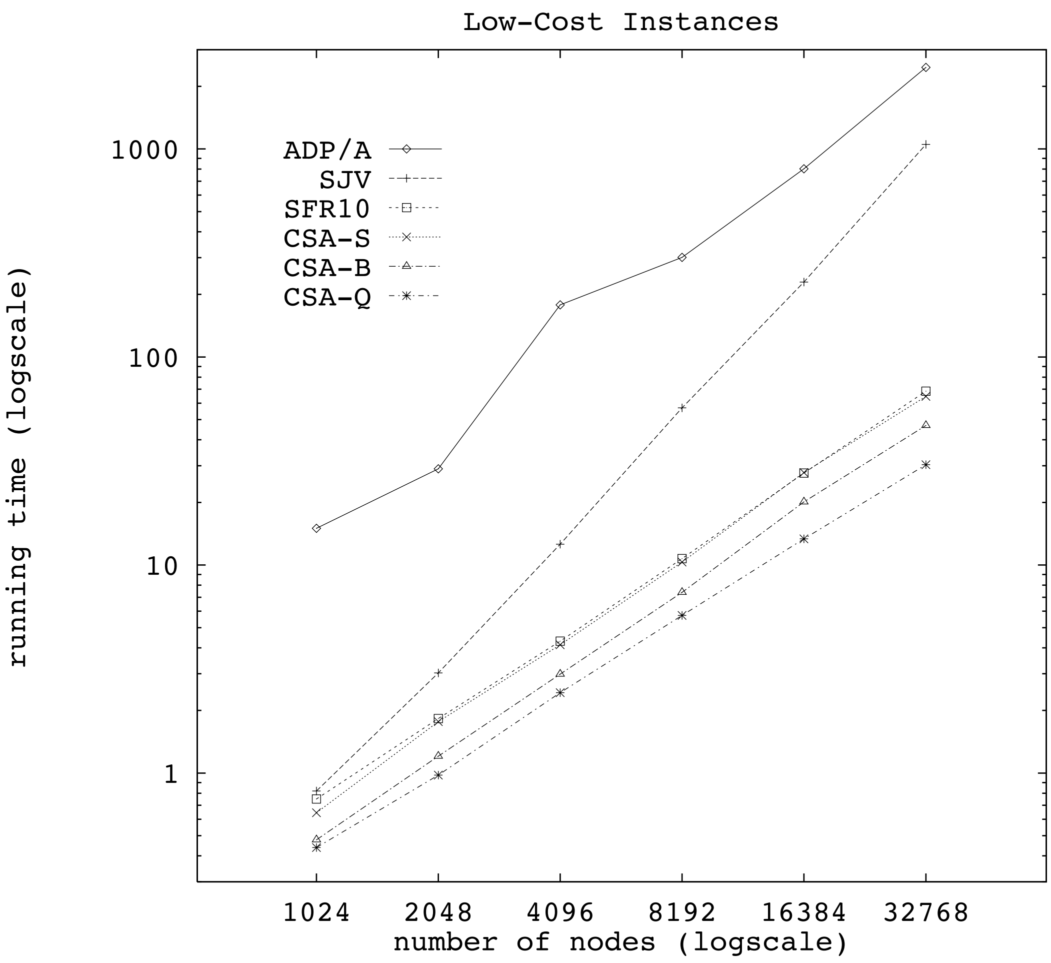
4.5.3 The Two-Cost Class

The two-cost data appear in Figure 4.6 and Figure 4.7. It is difficult for robust scaling algorithms to exploit the special structure of two-cost instances; the assignment problem for most of the graphs in this class amounts to finding a perfect matching on the high-cost edges, and none of the scaling codes we tested is able to take special advantage of this observation. Because SJV does not use scaling, it would seem a good candidate to perform especially well on this class, and indeed it does well on small two-cost instances. For large instances, however, SJV uses a great deal



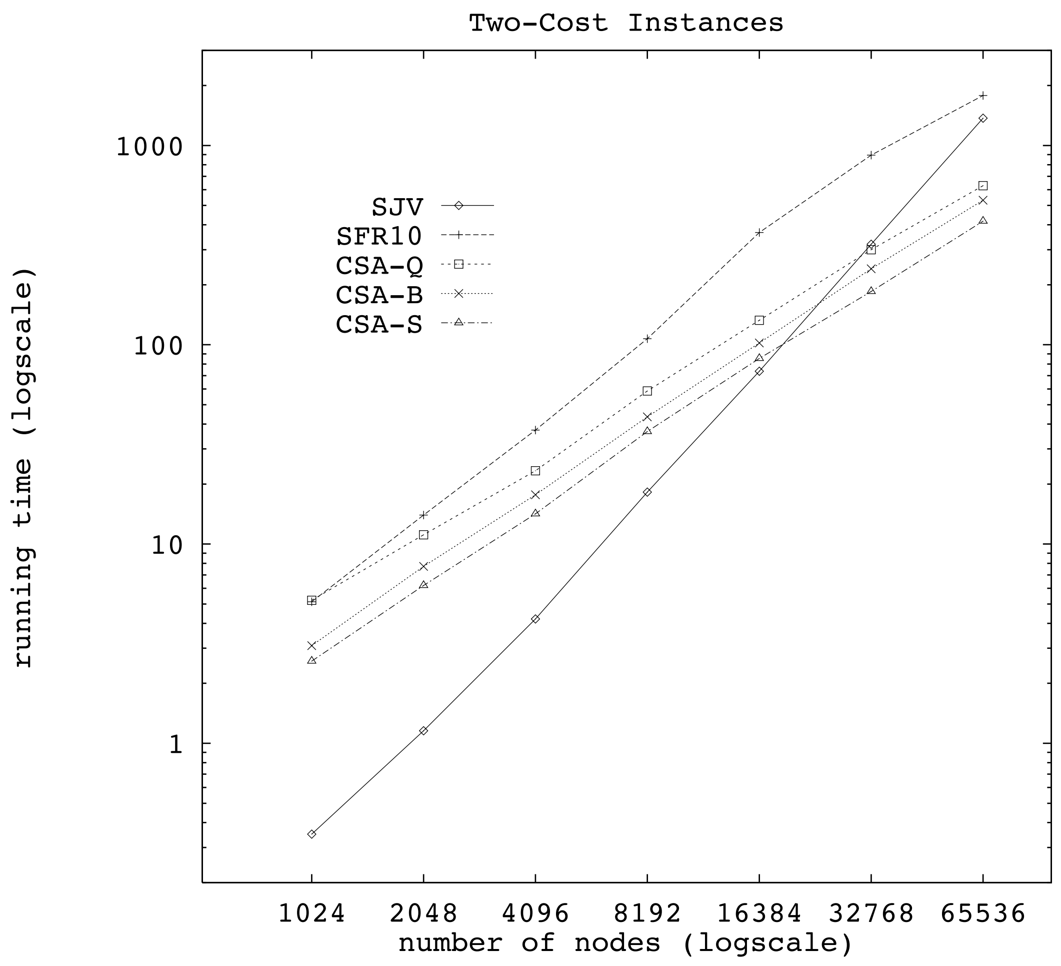
Nodes (|X|)	ADP/A time	SFR10 time	SJV time	CSA-B time	CSA-S time	CSA-Q time
1024	**17**	**1.2**	**0.87**	**0.7**	**1.1**	**0.5**
2048	**36**	**2.9**	**4.40**	**1.9**	**2.7**	**1.3**
4096	**132**	**6.4**	**18.1**	**4.3**	**6.2**	**2.8**
8192	**202**	**15.7**	**65.6**	**10.8**	**15.3**	**6.5**
16384	**545**	**37.3**	**266**	**25.5**	**38.3**	**14.3**
32768	**1463**	**85.7**	**1197**	**58.7**	**84.0**	**32.4**

Figure 4.4: Running Times for the High-Cost Class



Nodes ($\|X\|$)	ADP/A time	SFR10 time	SJV time	CSA-B time	CSA-S time	CSA-Q time
1024	**15**	**0.75**	**0.82**	**0.48**	**0.64**	**0.44**
2048	**29**	**1.83**	**3.03**	**1.21**	**1.77**	**0.98**
4096	**178**	**4.31**	**12.6**	**2.99**	**4.13**	**2.43**
8192	**301**	**10.7**	**57.0**	**7.39**	**10.3**	**5.72**
16384	**803**	**27.7**	**229**	**20.1**	**27.8**	**13.4**
32768	**2464**	**68.5**	**1052**	**46.9**	**64.6**	**30.3**

Figure 4.5: Running Times for the Low-Cost Class



Nodes	SFR10		SJV		CSA-B		CSA-S		CSA-Q	
($\|X\|$)	time	s	time	s	time	s	time	s	time	s
1024	**5.13**	0.09	**0.35**	0.00	**3.09**	0.24	**2.58**	0.15	**5.21**	0.33
2048	**14.0**	1.1	**1.16**	0.01	**7.72**	0.28	**6.19**	0.18	**11.1**	1.0
4096	**37.3**	1.1	**4.21**	0.16	**17.7**	1.1	**14.2**	1.6	**23.3**	2.4
8192	**107**	12	**18.2**	0.43	**43.4**	3.5	**36.7**	2.1	**58.6**	3.3
16384	**366**	81	**73.6**	0.58	**102**	2.8	**85.4**	3.2	**133**	7.8
32768	**894**	180	**320**	1.2	**240**	6.0	**185**	6.8	**299**	6.4
65536	**1782**	60	**1370**	5.8	**531**	15	**417**	11	**628**	25

Figure 4.6: Running Times (3-instance samples) for the Two-Cost Class

Nodes	SFR10		SJV		CSA-B		CSA-S		CSA-Q	
($\lvert X\rvert$)	time	s	time	s	time	s	time	s	time	s
1024	**5.05**	0.32	**0.35**	0.02	**3.07**	0.21	**2.56**	0.10	**4.93**	0.40
2048	**14.1**	1.1	**1.18**	0.04	**7.49**	0.37	**6.18**	0.26	**10.8**	0.73
4096	**37.4**	2.7	**4.22**	0.14	**17.7**	1.0	**14.7**	0.84	**24.1**	1.6
8192	**109**	9.8	**18.0**	0.37	**44.6**	2.5	**36.5**	1.5	**57.5**	3.2
16384	**314**	50	**73.7**	0.57	**105**	4.2	**84.1**	2.9	**130**	8.4
32768	**822**	194	**320**	2.1	**239**	8.5	**186**	4.8	**293**	15
65536	**2021**	342	**1376**	7.5	**524**	25	**426**	16	**637**	27

Figure 4.7: Running Times (15-instance samples) for the Two-Cost Class

of time in its shortest augmenting path phase, and performs poorly for this reason. Speculative arc fixing improves significantly upon the performance of the basic CSA implementation, and the kth-best heuristic hurts performance on this class of problems. It seems that the kth-best heuristic tends to speed up the last few iterations of *refine*, but it hurts in the early iterations. Like kth-best, the speculative arc fixing heuristic is able to capitalize on the fact that later iterations of *refine* can afford to ignore many of the arcs incident to each node, but by keeping all arcs of similar cost under consideration in the beginning, speculative arc fixing allows early iterations to run relatively fast. On this class, CSA-S is the winner, although for applications limited to this sort of strongly bimodal cost distribution, an unscaled push-relabel or blocking flow algorithm might perform better than any of the codes we tested. No running times are given in [48] for ADP/A on this problem class, but the authors suggest that their program performs very well on two-cost problems. Relative to those of the other codes, the running times of SFR10 are comparatively scattered at each problem size in this class; we believe this phenomenon results from the premature termination of early scaling phases in SFR10 (see Section 4.3).

The relatively large sample deviations shown in Figure 4.6 motivated our experiments with 15 instances of each problem size. The sample means and deviations of the 15-instance data are shown in Figure 4.7, and they are consistent with and very similar to the three-instance data shown in Figure 4.6.

4.5.4 The Fixed-Cost Class

Figure 4.8 gives the data for the fixed-cost problem class. On smaller instances of this class, CSA-B and CSA-Q have nearly the same performance. On instances with $|X| = 1024$ and $|X| = 2048$, CSA-Q is faster on fixed-cost problems than CSA-B, or indeed any of the other codes. On smaller instances, speculative arc fixing does not pay for itself; when $|X| = 2048$, the overhead is just paid for. Perhaps on larger instances, speculative arc fixing would pay off. It is doubtful, though, that CSA-S would beat CSA-Q on any instances of reasonable size. SJV exhibits the worst asymptotic behavior among the codes we tested on this problem class.

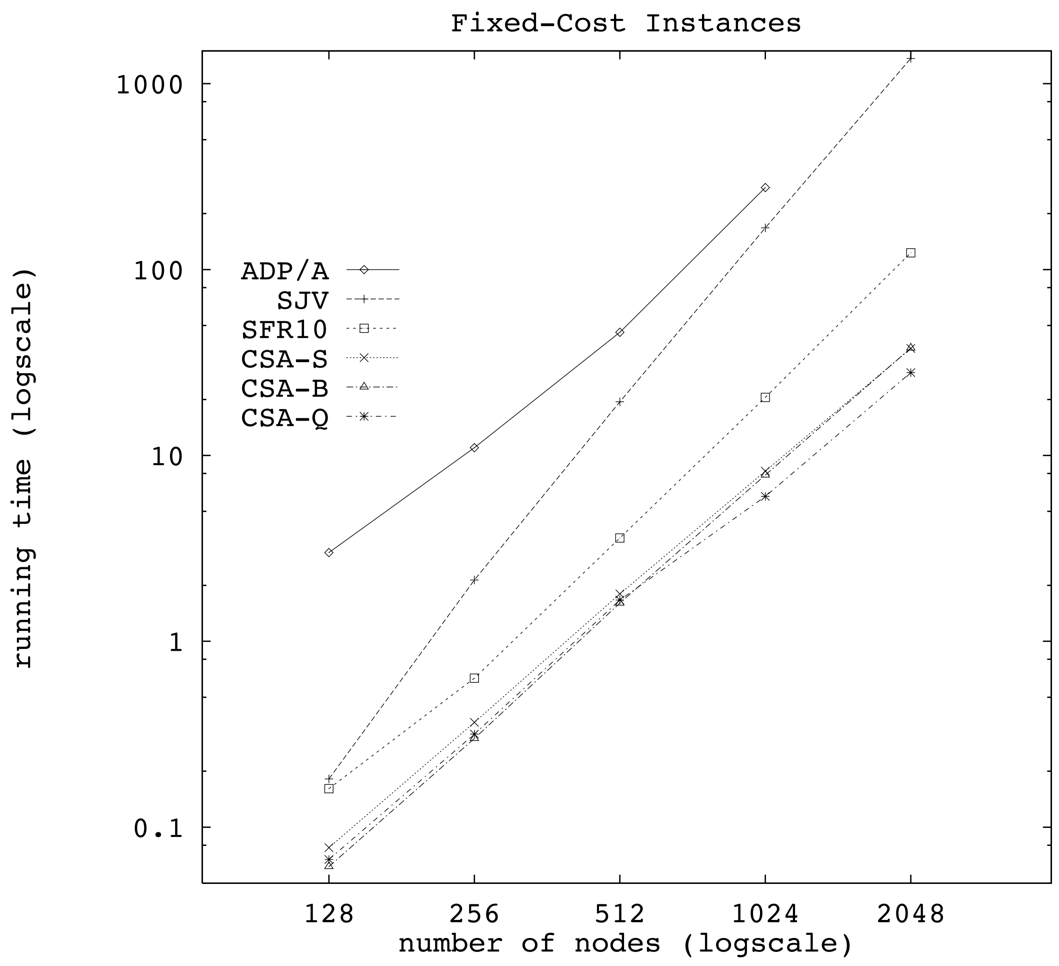
4.5.5 The Geometric Class

On geometric problems, both heuristics improve performance over the basic CSA-B code. The performance of CSA-S and CSA-Q is similar to and better than that of the other codes. The Jonker-Volgenant codes seem to have asymptotic behavior similar to the other codes on this class. See Figure 4.9.

Because the sample deviations shown in Figure 4.9 are somewhat large compared to those we observed on most other problem classes, we ran experiments on 15 instances as a check on the validity of the data. Statistics calculated over 15-instance samples are reported in Figure 4.10, and they are very much like the three-instance data.

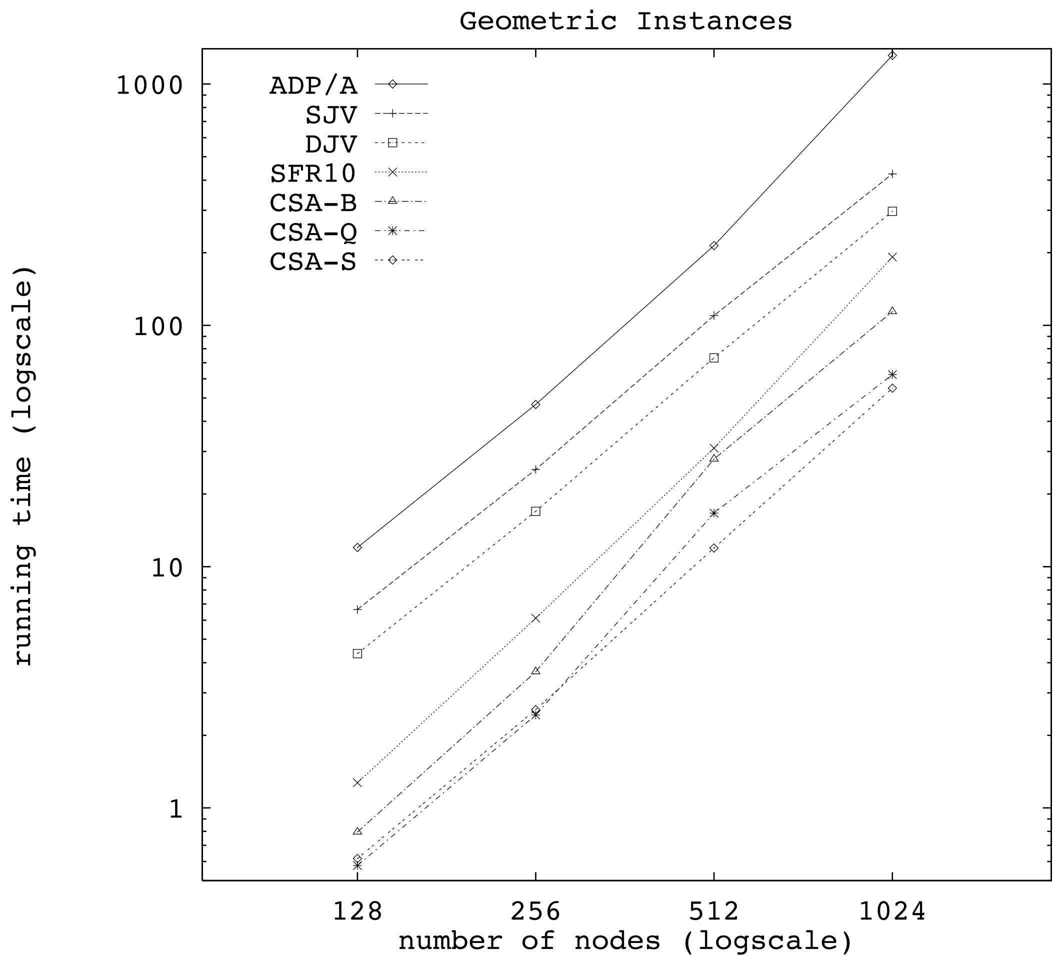
4.5.6 The Dense Class

The difference between Figures 4.10 and 4.11 shows that the codes' relative performance is significantly affected by changes in cost distribution. Except on very small instances, CSA-Q is the winner in this class; DJV is its closest competitor, with SJV performing fairly well also. As in the case of geometric problems, SJV and DJV seem to have asymptotic performance similar to the scaling and interior-point codes on this class.



Nodes (|X|)	ADP/A time	SFR10 time	SJV time	CSA-B time	CSA-S time	CSA-Q time
128	**3**	**0.16**	**0.18**	**0.06**	**0.08**	**0.07**
256	**11**	**0.63**	**2.14**	**0.30**	**0.37**	**0.32**
512	**46**	**3.59**	**19.4**	**1.6**	**1.8**	**1.7**
1024	**276**	**20.5**	**168**	**7.8**	**8.2**	**6.0**
2048	N/A	**123**	**1367**	**37.8**	**37.6**	**27.9**

Figure 4.8: Running Times for the Fixed-Cost Class



Nodes	ADP/A		SFR10		SJV		DJV		CSA-B		CSA-S		CSA-Q	
($\|X\|$)	time	s	time	s	time	s	time	s	time	s	time	s	time	s
128	**12**	0.5	**1.27**	0.46	**6.64**	4.4	**4.36**	2.9	**0.79**	0.28	**0.62**	0.05	**0.58**	0.19
256	**47**	1	**6.12**	0.23	**25.3**	3.3	**16.9**	2.0	**3.67**	0.67	**2.56**	0.08	**2.43**	0.34
512	**214**	42	**31.0**	4.1	**110**	2.8	**73.2**	1.0	**27.9**	8.1	**11.9**	0.89	**16.7**	3.7
1024	**1316**	288	**193**	19	**424**	51	**297**	32	**114**	24	**54.9**	1.42	**62.5**	2.6

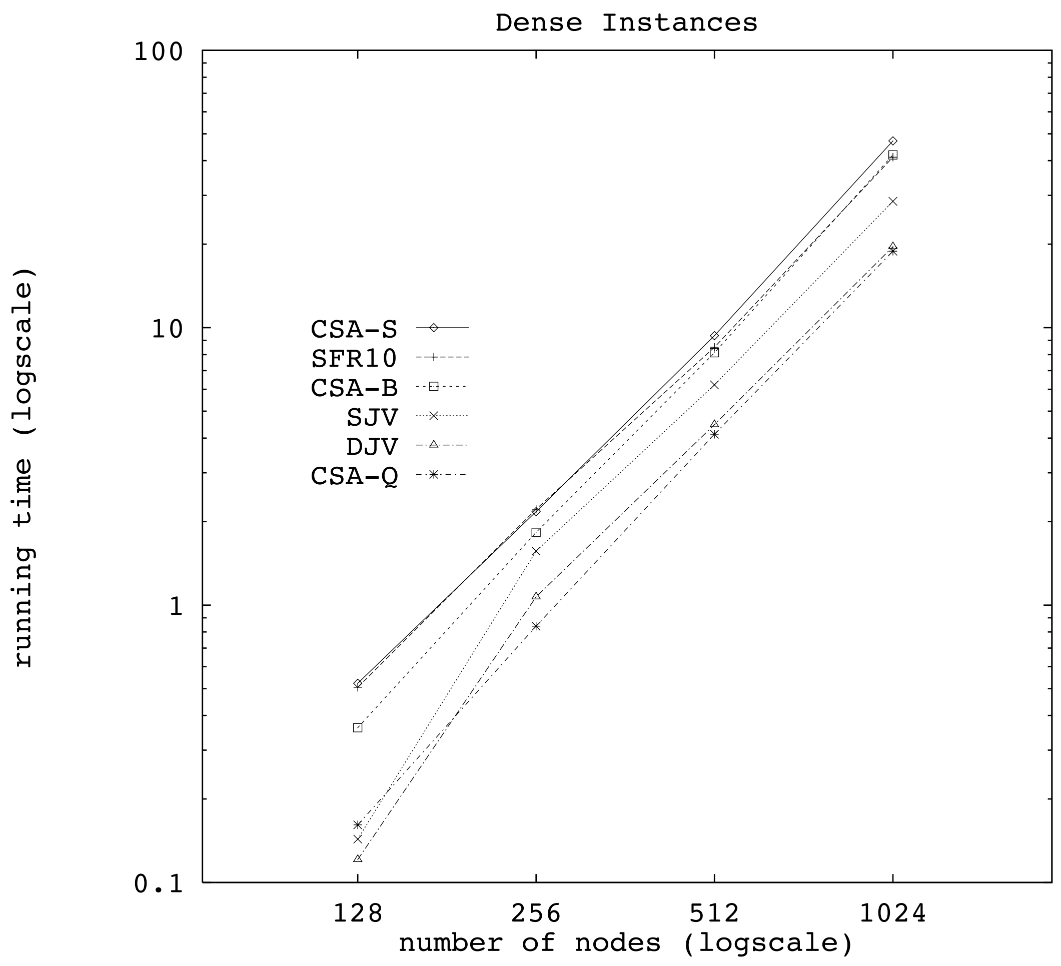
Figure 4.9: Running Times (3-instance samples) for the Geometric Class

Nodes	SFR10		SJV		DJV		CSA-B		CSA-S		CSA-Q	
($\|X\|$)	time	s	time	s	time	s	time	s	time	s	time	s
128	**1.28**	0.21	**5.96**	2.0	**3.85**	1.3	**0.78**	0.16	**0.61**	0.03	**0.57**	0.11
256	**6.21**	0.82	**26.1**	4.7	**17.5**	2.9	**3.72**	0.51	**2.63**	0.09	**2.50**	0.27
512	**35.0**	6.0	**101**	11	**68.2**	7.4	**23.2**	4.9	**11.8**	0.67	**15.1**	2.4
1024	**214**	54	**416**	38	**291**	25	**127**	27	**54.4**	2.2	**66.7**	9.7

Figure 4.10: Running Times (15-instance samples) for the Geometric Class

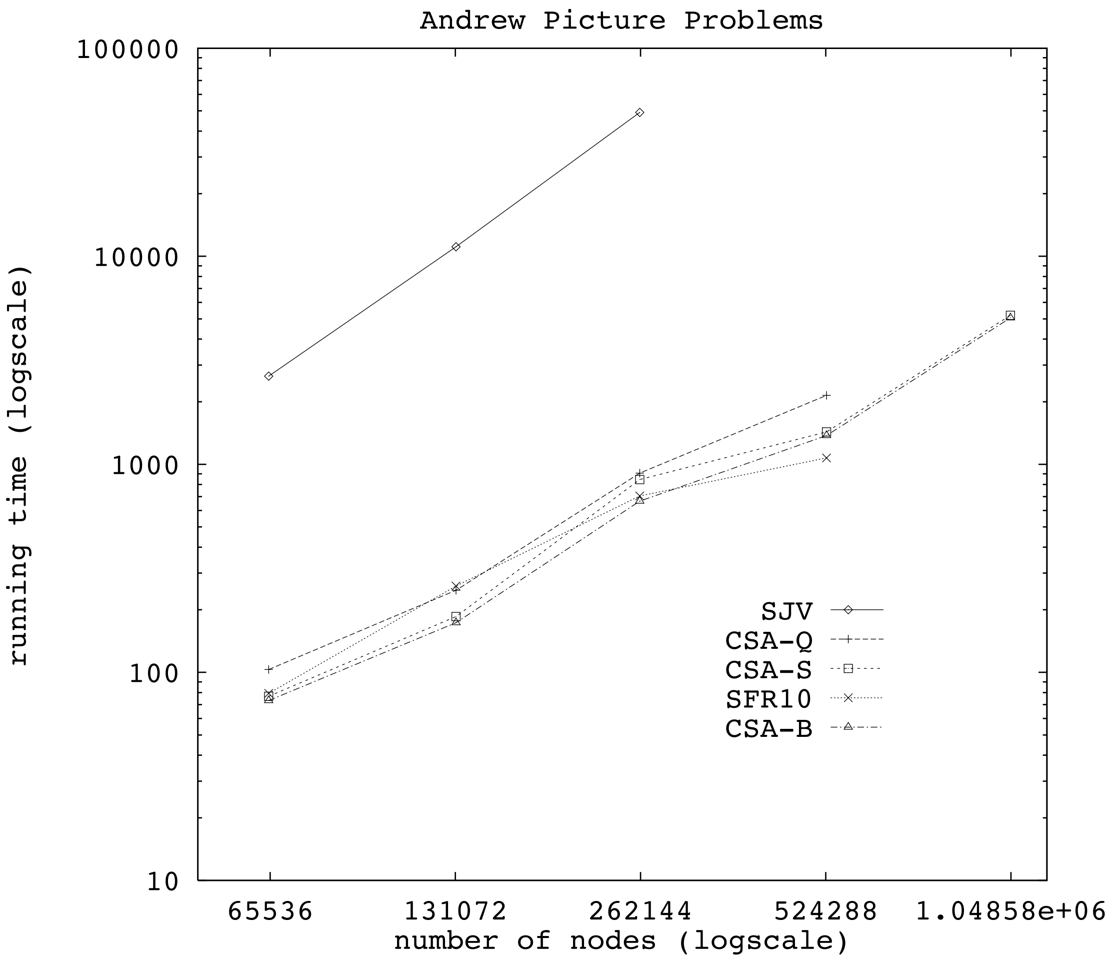
4.5.7 Picture Problems

Although the pictures used had very similar characteristics, the tentative conclusions we draw here about the relative performance of the codes seem to apply to a broader class of images. We performed trials on a variety of images generated and transformed by various techniques, and found no substantial differences in relative performance, although some pictures seem to yield more difficult assignment problems than others. On the picture problems we tried, SFR10 performs better than any of the CSA implementations; we believe that the "reverse-auction" phases performed by SFR10 [8] are critical to this performance difference. We were unable to obtain times for SJV and CSA-Q on the largest problem instance from each picture, nor from SFR10 on the largest problem instance from one of the pictures because the codes required too much memory. On the second-largest instance from each picture, our experiments suggested that SJV would require more than a day of CPU time, so we did not collect data for these cases. On picture problems CSA-Q performs significantly worse than either of the other two CSA implementations. This situation is no surprise because CSA-Q performs an additional pointer dereference almost every time it examines an arc. In such a sparse graph, the four arcs stored at each node exhaust the list of arcs incident to that node, so no benefit is to be had from the kth-best heuristic.



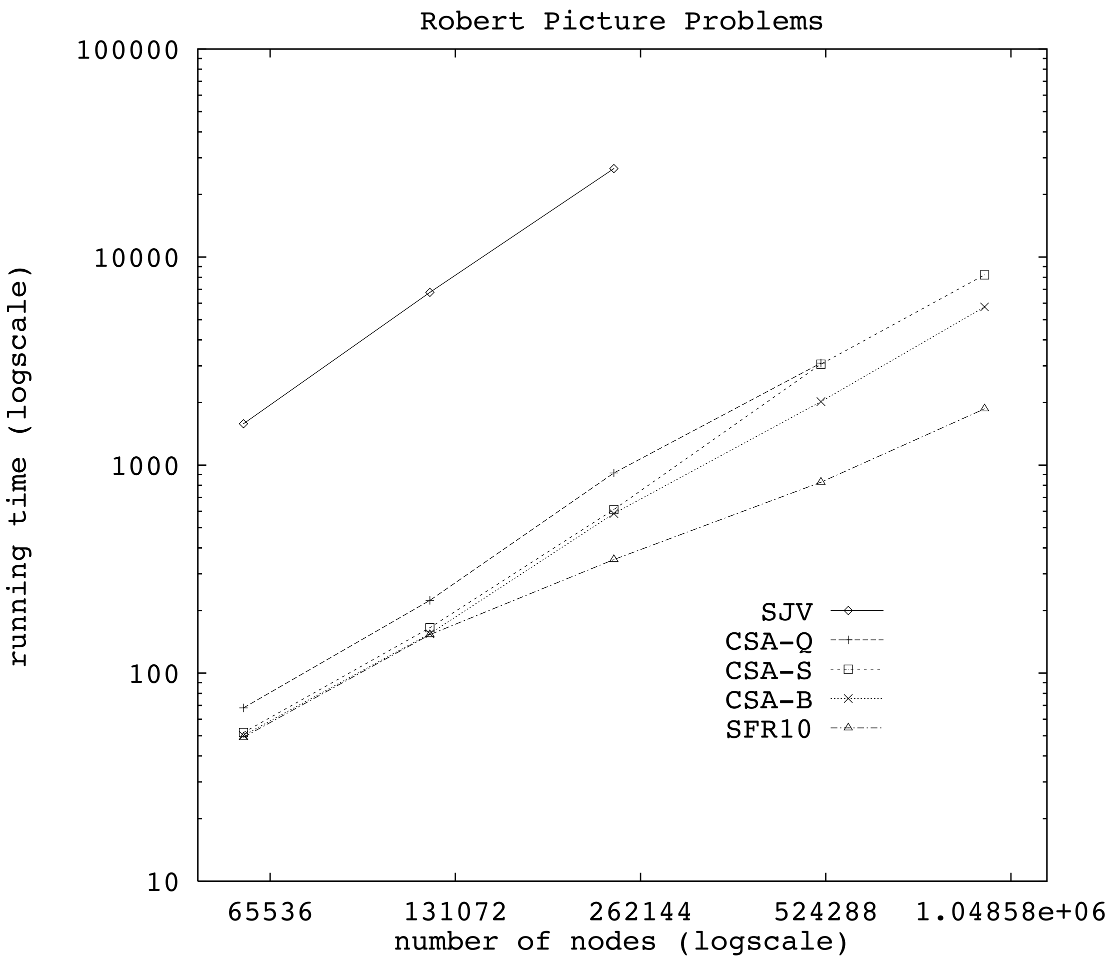
Nodes ($\|X\|$)	SFR10 time	SJV time	DJV time	CSA-B time	CSA-S time	CSA-Q time
128	**0.51**	**0.14**	**0.12**	**0.36**	**0.52**	**0.16**
256	**2.22**	**1.57**	**1.07**	**1.83**	**2.17**	**0.84**
512	**8.50**	**6.22**	**4.47**	**8.12**	**9.36**	**4.13**
1024	**41.2**	**28.5**	**19.6**	**42.0**	**47.1**	**18.9**

Figure 4.11: Running Times for the Dense Class



Nodes ($\|X\|$)	SFR10 time	SJV time	CSA-B time	CSA-Q time	CSA-S time
65158	**79.20**	**2656**	**73.23**	**103.3**	**76.70**
131370	**260.2**	**11115**	**173.2**	**248.0**	**185.5**
261324	**705.2**	**49137**	**665.1**	**907.8**	**844.8**
526008	**1073**	**N/A**	**1375**	**2146**	**1432**
1046520	**N/A**	**N/A**	**5061**	**N/A**	**5204**

Figure 4.12: Running Times for Problems from Andrew's Picture



Nodes ($\|X\|$)	SFR10 time	SJV time	CSA-B time	CSA-Q time	CSA-S time
59318	**49.17**	**1580**	**50.13**	**68.10**	**51.82**
119132	**153.1**	**6767**	**154.8**	**223.6**	**165.4**
237272	**351.4**	**26637**	**585.0**	**916.8**	**611.2**
515088	**827.8**	N/A	**2019**	**3095**	**3057**
950152	**1865**	N/A	**5764**	N/A	**8215**

Figure 4.13: Running Times for Problems from Robert's Picture

4.6 Concluding Remarks

Castañon [8] gives running times for an auction code called SF5 in addition to performance data for SFR10; SF5 and SFR10 are the fastest among the robust codes discussed. The data in [8] show that on several classes of problems, SF5 outperforms SFR10 by a noticeable margin. Comparing Castañon's reported running times for SFR10 with the data we obtained for the same code allows us to estimate roughly how SF5 performs relative to our codes. The data indicate that CSA-S and CSA-Q should perform at least as well as SF5 on all classes for which data are available, and that CSA-Q should outperform SF5 by a wide margin on some classes. A possible source of error in this technique of estimation is that Castañon reports times for test runs on cost-minimization problems, whereas all the codes we test here (including SFR10) are configured to maximize cost. The difference in every case is but a single line of code, but while on some classes minimization and maximization problems are similar, on other classes we observed that minimization problems were significantly easier for all the codes. This difference is unlikely to be a large error source, however, since the relative performance of the codes we tested was very similar for minimization problems and maximization problems.

It is interesting that SJV is asymptotically worse than all its competitors on every sparse class, and that SJV and DJV are asymptotically very similar to their competitors on the dense classes. DJV performs very well on the uniform dense problem class, but we feel SJV provides a more genuine reference point, since the other combinatorial codes could be sped up on dense problems by replacing their central data structures with an adjacency matrix representation similar to that in DJV.

From our tests and data from [48] and [8], we conclude that CSA-Q is a robust, competitive implementation that should be considered for use by those who wish to solve assignment problems in practice.

Chapter 5

Conclusions

The theory and implementation of combinatorial optimization algorithms have traditionally been viewed as orthogonal concerns. Interest in worst-case time bounds has often been set aside by practitioners seeking to develop practical codes to solve optimization problems. Even termination proofs have been sacrificed in some instances to achieve improvements in observed performance. Conversely, the theory of algorithms for optimization problems has developed a number of tools and techniques that provide good asymptotic behavior but that are so complicated to implement that they yield no practical advantage.

While much prior theoretical work has ignored issues of practical performance and most earlier implementation studies do not even mention the notion of worst-case performance, we have sought to bring the two approaches together to gain theoretical insight and improved codes simultaneously. We began our study with the global update heuristic, a technique known to play an important role in practical implementations, and we proved that the heuristic has attractive theoretical properties. We went on to use our theoretical understanding of cost-scaling push-relabel algorithms to develop an implementation that we feel is the best sequential code currently available to solve assignment problems.

It would have been most satisfying if the global update heuristic had participated in the best implementation we could devise. Then, theory and practice would have been united in a complete and appealing way. Unfortunately (at least from an aesthetic viewpoint), other heuristics gave better practical performance than global updates even though we could not prove the same bounds on their theoretical behavior. Nevertheless, the techniques that yield the best running times in practice owe their geneses to important theoretical insights without which they would probably not have been invented.

And so we close by noting that an important lesson of this work is that theory and practice in combinatorial optimization are an effective team. Theory helps explain observations made in practice and leads to improved implementations. Practical studies generate techniques such as global updates that have interesting theoretical properties and that lead to advances in our understanding of problem structure.

Appendix A

Generator Inputs

The assignment instances on which we ran our tests were generated as follows: Problems in the high-cost, low-cost, fixed-cost, and dense classes were generated using the DIMACS generator `assign.c`. Problems in the two-cost class were generated using `assign.c` with output post-processed by the DIMACS `awk` script `twocost.a`. Problems in the geometric class were generated using the DIMACS generator `dcube.c` with output post-processed by the DIMACS `awk` script `geomasn.a`. Picture problems were generated from images in the Portable Grey Map format using our program `p5pgmtoasn`. To obtain the DIMACS generators, connect to `dimacs.rutgers.edu` via anonymous `ftp`, or obtain the `csa` package (which includes the generators) as described below.

In each class except the picture class, we generated instances of various numbers of nodes N and using various seeds K for the random number generator. For each problem type and each N, either three or 15 values of K were used; the values were integers 270001 through 270003 or through 270015. For picture problems, we tested the codes on a single instance of each size.

A.1 The High-Cost Class

We generated high-cost problems using `assign.c` from the DIMACS distribution. The input parameters given to the generator are as follows, with the appropriate values substituted for N and K:

`nodes` N

`sources` $N/2$

`degree` $2 \log_2 N$

`maxcost` 100000000

`seed` K

A.2 The Low-Cost Class

Like high-cost problems, low-cost problems are generated using the DIMACS generator `assign.c`. The parameters to the generator are identical to those for high-cost problems, except for the maximum edge cost:

`nodes` N

`sources` $N/2$

`degree` $2 \log_2 N$

`maxcost` 100

`seed` K

A.3 The Two-Cost Class

Two-cost instances are derived from low-cost instances using the Unix `awk` program and the DIMACS `awk` script `twocost.a`. The instance with N nodes and seed K was generated using the following Unix command line, with input parameters identical to those for the low-cost problem class:

```
assign | awk -f twocost.a
```

A.4 The Fixed-Cost Class

We generated fixed-cost instances using `assign.c`, with input parameters as follows:

`nodes` N

`sources` $N/2$

`degree` $N/16$

`maxcost` 100

`multiple`

`seed` K

A.5 The Geometric Class

We generated geometric problems using the DIMACS generator `dcube.c` and the DIMACS `awk` script `geomasn.a`. We gave input parameters to `dcube` as shown below, and used the following Unix command line:

```
dcube | awk -f geomasn.a
```

`nodes` N

`dimension` 2

`maxloc` 1000000

`seed` K

A.6 The Dense Class

We generated dense problems using `assign.c`, with input parameters as follows:

`nodes` N

`sources` $N/2$

`complete`

`maxcost` 1000000

`seed` K

Appendix B

Obtaining the CSA Codes

To obtain a copy of the CSA codes, DIMACS generators referred to in this thesis, and documentation files, send mail to `ftp-request@theory.stanford.edu` and use `send csas.tar` as the subject line; you will automatically be mailed a `uuencoded` copy of a `tar` file.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[2] R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.

[3] R. S. Barr, F. Glover, and D. Klingman. An improved version of the out-of-kilter method and a comparative study of computer codes. *Math. Prog.*, 7:60–86, 1974.

[4] D. P. Bertsekas. Distributed asynchronous relaxation methods for linear network flow problems. In *Proc. 25th IEEE Conference on Decision and Control, Athens, Greece*, 1986.

[5] D. P. Bertsekas. The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem. *Annals of Oper. Res.*, 14:105–123, 1988.

[6] D. P. Bertsekas. *Linear Network Optimization: Algorithms and Codes.* MIT Press, 1991.

[7] R. G. Bland, J. Cheriyan, D. L. Jensen, and L. Ladańyi. An Empirical Study of Min Cost Flow Algorithms. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 119–156. AMS, 1993.

[8] D. A. Castañon. Reverse Auction Algorithms for Assignment Problems. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 407–430. AMS, 1993.

[9] B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. In *Proc. 4th Integer Prog. and Combinatorial Opt. Conf.*, pages 157–171, 1995.

[10] V. Chvátal. *Linear Programming.* W. H. Freeman and Company, New York, 1983.

[11] W. H. Cunningham. A Network Simplex Method. *Math. Programming*, 11:105–116, 1976.

[12] G. B. Dantzig. *Linear Programming and Extensions.* Princeton Univ. Press, Princeton, NJ, 1962.

[13] G. B. Dantzig, L. R. Ford, and D. R. Fulkerson. A Primal-Dual Algorithm for Linear Programs. In H. W. Kuhn and A. W. Tucker, editors, *Linear Inequalities and Related Systems.* Princeton Univ. Press, Princeton, NJ, 1956.

[14] 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.

[15] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.

[16] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[17] W. L. Eastman. *Linear Programming with Pattern Constraints.* PhD thesis, Harvard University, 1958. (Also available as Report No. BL. 20, The Computation Laboratory, Harvard University, 1958).

[18] S. Even and R. E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[19] T. Feder and R. Motwani. Clique Partitions, Graph Compression and Speeding-up Algorithms. In *Proc. 23st Annual ACM Symposium on Theory of Computing*, pages 123–133, 1991.

[20] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[21] S. Fujishige, K. Iwano, J. Nakano, and S. Tezuka. A Speculative Contraction Method for the Minimum Cost Flows: Toward a Practical Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 219–246. AMS, 1993.

[22] H. N. Gabow and R. E. Tarjan. Almost-Optimal Speed-ups of Algorithms for Matching and Related Problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 514–527, 1988.

[23] H. N. Gabow and R. E. Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, pages 1013–1036, 1989.

[24] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[25] A. V. Goldberg. An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. In *Proc. 3rd Integer Prog. and Combinatorial Opt. Conf.*, pages 251–266, 1993.

[26] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan. The Use of Dynamic Trees in a Network Simplex Algorithm for the Maximum Flow Problem. *Math. Prog.*, 50:277–290, 1991.

[27] A. V. Goldberg and R. Kennedy. An Efficient Cost Scaling Algorithm for the Assignment Problem. Technical Report STAN-CS-93-1481, Department of Computer Science, Stanford University, 1993. Math. Programming, to appear.

[28] A. V. Goldberg and R. Kennedy. Global Price Updates Help. Technical Report STAN-CS-94-1509, Department of Computer Science, Stanford University, 1994.

[29] A. V. Goldberg and M. Kharitonov. On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 157–198. AMS, 1993.

[30] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-Time Parallel Algorithms for Matching and Related Problems. *J. Algorithms*, 14:180–213, 1993.

[31] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[32] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Successive Approximation. *Math. of Oper. Res.*, 15:430–466, 1990.

[33] D. Goldfarb and M. D. Grigoriadis. A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow. *Annals of Oper. Res.*, 13:83–123, 1988.

[34] M. D. Grigoriadis. An Efficient Implementation of the Network Simplex Method. *Math. Prog. Study*, 26:83–111, 1986.

[35] P. Hall. On Representatives in Subsets. *J. Lond. Math. Soc.*, 10:26–30, 1935.

[36] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut of a Graph. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1992.

[37] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[38] D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: F1rst DIMACS Implementation Challenge.* AMS, 1993.

[39] R. Jonker and A. Volgenant. A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems. *Computing*, 38:325–340, 1987.

[40] A. V. Karzanov. O nakhozhdenii maksimal'nogo potoka v setyakh spetsial'nogo vida i nekotorykh prilozheniyakh. In *Matematicheskie Voprosy Upravleniya Proizvodstvom*, volume 5. Moscow State University Press, Moscow, 1973. In Russian; title translation: On Finding Maximum Flows in Network with Special Structure and Some Applications.

[41] A. V. Karzanov. Tochnaya otzenka algoritma nakhojdeniya maksimalnogo potoka, primenennogo k zadache "o predstavitelyakh". In *Problems in Cibernetics*, volume 5, pages 66–70. Nauka, Moscow, 1973. In Russian; title translation: The exact time bound for a maximum flow algorithm applied to the set representatives problem.

[42] D. Knuth. Personal communication. 1993.

[43] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

[44] J. M. Mulvey. Pivot strategies for primal-simplex network codes. *J. Assoc. Comput. Mach.*, 25:266–270, 1978.

[45] Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.

[46] J. B. Orlin and R. K. Ahuja. New Scaling Algorithms for the Assignment and Minimum Cycle Mean Problems. *Math. Prog.*, 54:41–56, 1992.

[47] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[48] K. G. Ramakrishnan, N. K. Karmarkar, and A. P. Kamath. An Approximate Dual Projective Algorithm for Solving Assignment Problems. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 431–452. AMS, 1993.

[49] A. Schrijver. *Theory of Linear and Integer Programming.* J. Wiley & Sons, 1986.

[50] É. Tardos. A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica*, 5(3):247–255, 1985.

[51] R. E. Tarjan. Efficiency of the Primal Network Simplex Algorihm for the Minimum-Cost Circulation Problem. Technical Report CS-TR-187-88, Department of Computer Science, Princeton University, 1988.