# Efficient Graph Algorithms for Sequential and Parallel Computers

by

Andrew Vladislav Goldberg

B. S., Massachusetts Institute of Technology
(1982)
M. S., University of California at Berkeley
(1983)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements of the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

January 1987



Signature of Author: _____
Department of Electrical Engineering and Computer Science
January 30, 1987

Certified by: _____
Charles E. Leiserson
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: _____
Arthur C. Smith
Chairman, Department Committee

EFFICIENT GRAPH ALGORITHMS
FOR
SEQUENTIAL AND PARALLEL COMPUTERS

by

Andrew Vladislav Goldberg

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the requirements for the Degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

## Abstract

In this thesis we study graph algorithms, both in sequential and parallel contexts. In the following outline of the thesis, algorithm complexities are stated in terms of the number of vertices $n$, the number of edges $m$, the largest absolute value of capacities $U$, and the largest absolute value of costs $C$.

In Chapter 1 we introduce a new approach to the maximum flow problem that leads to better algorithms for the problem. These algorithms include an $O(nm\log(n^2/m))$ time sequential algorithm, an $O(n^2\log n)$ time parallel algorithm that uses $O(n)$ processors and $O(m)$ memory, and both synchronous and asynchronous distributed algorithms.

Chapter 2 is devoted to the minimum cost flow problem, which is a generalization of the maximum flow problem. We introduce a framework that allows the generalization the of maximum flow techniques to the minimum-cost flow problem. This framework allows us to design effecient algorithms for the minimum-cost flow problem. We exhibit $O(nm\log(n)\log(nC))$, $O(n^{5/3}m^{2/3}\log(nC))$, and $O(n^3\log(nC))$ time sequential algorithms as well as parallel and distributed algorithms.

In Chapter 3 we address implementation of parallel algorithms through a case-study of an implementation of a parallel maximum flow algorithm. Parallel prefix operations play an important role in our implementation. We present experimental results acheived by the implementation.

Parallel symmetry-breaking techniques are the main topic of Chapter 4. We give an $O(\lg^* n)$ algorithm for 3-coloring a rooted tree. This algorithm is used to improve several parallel algorithms, including algorithms for $\Delta+1$-coloring and finding maximal independent set in constant-degree graphs, 5-coloring planar graphs, and finding a maximal matching in planar graphs. We also prove lower bounds on the parallel complexity of the maximal independent set problem and the problem of 2-coloring a rooted tree.

**Thesis Supervisor**: Professor Charles E. Leiserson

**Title**: Associate Professor of Computer Science and Engineering

# Contents

# Acknowledgments

I would like to thank all the individuals who have contributed to the contents of this thesis and to my professional development in general. I am very grateful to my advisor, Charles Leiserson, who taught me the fundamentals of parallel computation and offered encouragement and support during the course of this research. His help on both technical and nontechnical matters was of the quality that only an excellent advisor can provide.

During the course of my research, I had the great pleasure of cooperating with Bob Tarjan. I am very grateful for his numerous suggestions and for detailed comments on a draft of this thesis. In the course of my graduate education, I profited from many discussions with David Shmoys and I would like to thank him for his contributions to my research. My research has also benefited from the joint work with Serge Plotkin, whose comments were very helpful.

I would like to thank Tom Leighton and Ron Rivest for serving on my thesis committee. Their suggestions have improved both the presentation and the technical contents of this thesis. Ron Rivest also helped me to achieve high standards of academic scholarship.

I also would like to take this opportunity to thank Michael Sipser for getting me excited about computer science, Gene Lawler for introducing me to the area of combinatorial optimization, Dick Karp for supervising my first research steps, and Karl Lieberherr for contributing to my further professional development.

I have also benefited from comments and suggestions of Baruch Awerbuch, Robert Bland, and Jim Orlin.

I am grateful to the Fannie and John Hertz Foundation which provided me with a fellowship during my graduate education. I would like to thank Lowell Wood of the foundation for his advice and encouragement. Additional support was provided

by the Defense Advanced Research Projects Agency under contract #N00014-80-C-0622.

Finally, I would like to thank Thinking Machines Corporation for the opportunity to spend a summer in the wonderful environment of this company and for the providing an access to a Connection Machine for my experimental work.

# Introduction

Advances in computing technology have enabled computers to solve a wider class of problems. One way to extend this class of problems is to build more powerful computers. Another way is to design better algorithms. Both approaches have their advantages and disadvantages. Fast hardware is essential for many applications, such as these that require real-time computations. Furthermore, any program runs faster on a faster machine. On the other hand, if an algorithm takes too much time to solve real-life problems due to high (e.g., exponential) running time, the combinatorial explosion will prevent the algorithm from solving these problems even if computers become thousands of times faster.

Parallel computers are one of the most promising recent developments in high-performance hardware. Parallel machines with tens of thousands processors are already available commercially, and machines with millions of processors can be built with today's technology. The class of problems for which efficient parallel algorithms are known, however, is much smaller then the class of problems for which efficient sequential algorithms are known. The implementation of parallel algorithms is also not as well understood.

This thesis addresses the issue of designing efficient sequential and parallel algorithms for graph-theoretic problems. These problems are important because of their applications, both outside of and in the field of computer science. For this reason, graph algorithms have become one of the most studied fields of theoretical computer science. We shall see, however, that even algorithms for the classical problems in the area can be significantly improved.

This thesis has four chapters. In Chapter 1 we study the maximum flow problem, a classical problem in the theory of network flows. We introduce a new approach to the problem and use this this approach to design better sequential, parallel, and

distributed algorithms for the problem. In particular, we exhibit $O(nm \log(n^2/m))$ sequential algorithm for the problem, improving the previous upper bound. A parallel version of our algorithm runs in $O(n^2 \log n)$ time using $O(n)$ processors and $O(m)$ memory. These time and processor bounds are as good as for those the best of the previously known algorithms, and the memory bound is better, making our algorithm much more practical. We also exhibit and analyze synchronous and asynchronous distributed algorithms.

In Chapter 2 we study another classical problem from the theory of network flows: the minimum-cost flow problem. We show how maximum flow techniques (including the techniques introduced in Chapter 1) can be extended to the minimum-cost flow problem. We introduce a new framework for the minimum-cost flow problem and exhibit algorithms that significantly improve the known complexity bounds. Our sequential algorithms achieve $O(\min(nm \log n, n^{5/3}m^{2/3}, n^3) \log(nC))$ running time. A parallel version of our algorithm achieves $O(n^2 \log(n) \log(nC))$ running time using $O(n)$ processors and $O(n^2)$ memory. A different parallel algorithm uses only $O(m)$ memory and $O(n)$ processors. The parallel running time bound we can prove for this algorithm is $O(n^3 \log(n) \log(nC))$; however, we conjecture that the running time is in fact $O(n^2 \log(n) \log(nC))$.

How does one implement parallel algorithms? Are parallel algorithms really faster then the sequential algorithms? These questions are addressed in Chapter 3, where we describe a parallel implementation of our maximum flow algorithm that uses parallel prefix operations as primitives. In this chapter we also present experimental results achieved by the implementation.

Our study of design and implementation of parallel network flow algorithms motivates the search for general techniques for the design of parallel algorithms. Chapter 4 introduces efficient symmetry-breaking techniques which are very important in the context of parallel computation. We show how to apply these techniques to design better parallel algorithms. Our main result is an $O(\lg^* n)$ parallel time algorithm to 3-color a rooted tree. This algorithm is used to improve parallel algorithms for several graph-theoretic problems, including the problem of 5-coloring a

planar graph, the problem of $(\Delta + 1)$-coloring and finding a maximal independent set in a constant-degree graph, and a few other problems. All these algorithms use a linear number of processors. We also show that the 2-coloring of a rooted tree requires $\Omega(\log n / \log\log n)$ time if a polynomial number of processors is used, and prove the same lower bound for the maximal independent set problem on general graphs.

# Chapter 1

# The Maximum Flow Problem

## 1.1 Introduction

In this chapter we introduce a new approach to the maximum flow problem and use this approach to design better sequential and parallel algorithms for the problem.

The problem of finding a maximum flow in a directed graph with edge capacities arises in many settings in operations research and other fields, and efficient algorithms for the problem have received a great deal of attention. Extensive discussion of the problem and its applications can be found in the books of Ford and Fulkerson [19], Lawler [52], Even [16], Papadimitriou and Steiglitz [60], and Tarjan [71]. Table 1.1 summarizes polynomial-time algorithms for the problem. Time bounds are stated in terms of the number $n$ of vertices, the number $m$ of edges, and in two cases in terms of an upper bound $U$ on the edge capacities (assumed in these cases to be integers).

The first maximum flow algorithm, due to Ford and Fulkerson [20], works by finding augmenting paths. The algorithms in the table are variations of the Ford-Fulkerson algorithm, incorporating the observation of Edmonds and Karp [15] that augmenting along shortest paths leads to a polynomial-time algorithm (algorithm 1). To further improve the efficiency, Dinic [14] proposed a method to find all shortest augmenting paths in one phase. Algorithms 2-11 use Dinic's method. Al-

| # | Date | Discoverer | Running Time | References |
|---|---|---|---|---|
| 1 | 1969 | Edmonds and Karp | $O(nm^2))$ | [15] |
| 2 | 1970 | Dinic | $O(n^2m))$ | [14] |
| 3 | 1974 | Karzanov | $O(n^3)$ | [47] |
| 4 | 1977 | Cherkasky | $O(n^2m^{1/2})$ | [12] |
| 5 | 1978 | Malhotra, Pramodh Kumar, and Maheshwari | $O(n^3)$ | [56] |
| 6 | 1978 | Galil | $O(n^{5/3}m^{2/3})$ | [27] |
| 7 | 1978 | Galil and Naamad; Shiloach | $O(nm(\log n)^2)$ | [29] [64] |
| 8 | 1980 | Sleator and Tarjan | $O(nm \log n)$ | [67,68] |
| 9 | 1982 | Shiloach and Vishkin | $O(n^3)$ | [66] |
| 10 | 1983 | Gabow | $O(nm \log U)$ | [26] |
| 11 | 1984 | Tarjan | $O(n^3)$ | [72] |
| 12 | 1985 | Goldberg | $O(n^3)$ | [33] |
| 13 | 1986 | Goldberg and Tarjan | $O(nm \log(n^2/m))$ | [37] |
| 14 | 1986 | Ahuja and Orlin | $O(nm + n^2 \log U)$ | [1] |

Table 1.1: Polynomial-time algorithms for the maximum flow problem. Algorithm 13 is presented in this chapter.

gorithms 12-14 are based on the approach described in this paper.

There is no clear winner among the algorithms in the table that are based on the Dinic's method. Algorithms 3, 5, 9 and 11 are designed to be fast on dense graphs, and algorithms 4, 6, 7, 8, and 10 are designed to be fast on sparse graphs. For dense graphs, the best known bound of $O(n^3)$ was first obtained by Karzanov [47]; Malhotra, Pramodh Kumar, and Maheshwari [56] and Tarjan [72] have given simpler $O(n^3)$-time algorithms. For sparse graphs, Sleator and Tarjan's bound of $O(nm \log n)$ [67,68] is the best to date. For a small range of densities ($m$ between $\Omega(n^2/(\log n)^3)$ and $O(n^2)$), Galil's bound of $O(n^{5/3}m^{2/3})$ [27] is best. For sparse graphs with integer edge capacities of moderate size, Gabow's scaling algorithm [26] is best. Among the algorithms in the table, the only parallel algorithm is that of Shiloach and Vishkin [66]. This algorithm has a parallel running time of $O(n^2 \log n)$ but requires $O(nm)$ space. Vishkin (private communication) has improved the space bound to $O(n^2)$. Our work has been motivated by the Shiloach-Vishkin algorithm.

In this chapter we present a different approach to the maximum flow problem, which is the basis for algorithms 12-14 in the table. Our method uses Karzanov's idea of a *preflow*. A preflow is like a flow except that the total amount flowing into a vertex can exceed the total amount flowing out. During each phase, Karzanov's algorithm maintains a preflow in an acyclic network. The algorithm pushes flow through the network to find a blocking flow, which determines the acyclic network for the next phase. Our algorithm abandons the idea of finding a flow in each phase, and also abandons the idea of global phases. Instead, our algorithm maintains a preflow in the original network and pushes local flow excess toward the sink along what it estimates to be shortest paths in the residual graph. This pushing of flow changes the residual graph and paths to the sink may become saturated. Excess that cannot be moved to the sink is returned to the source, also along estimated shortest paths. Only when the algorithm terminates does the preflow become a flow, and then it is a maximum flow.

The algorithm is simple and intuitive. It has natural implementations in sequential and parallel models of computation. We present a simple sequential implementation that runs in $O(n^3)$ time and a more complicated sequential implementation that uses the dynamic tree data structure of Sleator and Tarjan [68,69,71] and runs in $O(nm \log(n^2/m))$ time. The latter bound matches the best known bounds as a function of $n$ and $m$ for both sparse and dense graphs and is better than known bounds on graphs of intermediate density. We present a parallel version of the algorithm running in $O(n^2 \log n)$ time using $O(1)$ words of storage per edge. This matches the time bound of the Shiloach-Vishkin algorithm, but our improved space bound allows implementation on a model of distributed computation in which the amount of space per processor at a vertex is bounded by the vertex degree. Recently, Ahuja and Orlin [1] used the approach described in this chapter to develop an $O(nm + n^2 \log U)$ algorithm for the problem, improving Gabow's bound of $O(nm \log U)$ [26].

This chapter contains six sections in addition to the introduction. Section 1.2 describes a generic version of the algorithm. Section 1.3 proves its termination and correctness. Section 1.4 refines the algorithm to produce an $O(n^3)$-time sequential

implementation. Section 1.5 adds the use of dynamic trees and thereby improves the sequential time bound to $O(nm \log(n^2/m))$. Section 1.6 discusses efficient distributed and parallel implementations. Section 1.7 contains some concluding remarks and open problems.

The approach presented in this section has been pioneered by the author [33]. The version presented here generalizes and improves the original results. These improvements represent joint work with Tarjan [37].

## 1.2 A Generic Maximum Flow Algorithm

Let $G = (V, E)$ be a directed graph with vertex set $V$ and edge set $E$. We shall denote the size of $V$ by $n$ and the size of $E$ by $m$. For ease in stating time bounds we assume $m \geq n - 1 \geq 4$. For a pair of vertices $v$ and $w$ we define the *distance* $d_G(v, w)$ from $v$ to $w$ in $G$ to be the minimum number of edges on a path from $v$ to $w$ in $G$; if there is no such path, we assume $d_G(v, w) = \infty$. A graph $G = (V, E)$ is a *flow network* if it has two distinguished vertices, a *source* $s$ and a *sink* $t$, and a positive real-valued *capacity* $u(v, w)$ for each edge $(v, w) \in E$. We extend the capacity function to all vertex pairs by defining $u(v, w) = 0$ if $(v, w) \notin E$. A *flow* $f$ on $G$ is a real-valued function on vertex pairs satisfying the following constraints:

$$f(v, w) \leq u(v, w) \text{ for all } (v, w) \in V \times V \quad \text{(capacity constraint)}, \tag{1.1}$$

$$f(v, w) = -f(w, v) \text{ for all } (v, w) \in V \times V \quad \text{(antisymmetry constraint)}, \tag{1.2}$$

$$\sum_{w \in V} f(v, w) = 0 \text{ for all } v \in V - \{s, t\} \quad \text{(flow conservation constraint)}. \tag{1.3}$$

*Remark*: The antisymmetry constraint (1.2), which is nonstandard, has two purposes: (i) it eliminates the possibility of having positive flow on both edges of an opposing pair $(v, w)$ and $(w, v)$, a possibility that creates certain technical difficulties, and (ii) it simplifies the formal expression of constraints such as the flow

conservation constraint (1.3). To gain an intuition one should think only of the positive part of the flow function; the appropriate interpretation of the flow conservation constraint is that the total flow into any vertex $v \notin \{s,t\}$ equals the total flow out of $v$.

The *value* $|f|$ of a flow $f$ is the net flow into the sink:

$$|f| = \sum_{v \in V} f(v,t).$$

A *maximum flow* is a flow of maximum value.

The problem we wish to solve is that of computing a maximum flow in a given network. Our algorithm solves this problem by manipulating a *preflow* $f$ on the network. A preflow is a real-valued function on vertex pairs satisfying (1.1) and (1.2) above, as well as the following weakened form of 1.3:

$$\sum_{i \in V} f(i,v) \geq 0 \text{ for all } v \in V - \{s\} \quad \text{(nonnegativity constraint)}. \tag{1.4}$$

That is, the total flow into any vertex $v \neq s$ is at least as great as the total flow out of $v$. We define the *flow excess* $e(v)$ of a vertex $v$ to be $\sum_{i \in V} f(i,v)$, the net flow into $v$.

The preflow algorithm works by examining vertices other than $s$ and $t$ with positive flow excess and pushing excess from them to vertices estimated to be closer to the sink $t$, with the goal of getting as much excess as possible to $t$. If the sink is not reachable from a vertex with a positive excess, however, the algorithm pushes this excess to vertices estimated to be closer to the source $s$. Eventually the algorithm reaches a state in which all vertices other than $s$ and $t$ have zero excess. At this point the preflow $f$ is a flow; in fact, $f$ is a maximum flow.

Before describing the algorithm, we first address two issues: how to move flow excess from one vertex to another, and how to estimate the distances from a vertex to $s$ or to $t$.

To deal with the first issue, we define the *residual capacity* $r_f(v,w)$ of a vertex pair $(v,w)$ to be $u(v,w) - f(v,w)$. If vertex $v$ has positive excess and pair $(v,w)$ has positive residual capacity, then an amount of flow excess up to $\delta = \min(e(v), r_f(v,w))$ can be moved from $v$ to $w$ by adding $\delta$ to $f(v,w)$ (and subtracting $\delta$ from $f(w,v)$). Observe that there are two ways a pair $(v,w)$ can have positive residual capacity: either $(v,w)$ is an edge with flow less than its capacity (edge $(v,w)$ is said to be *unsaturated*), or $(w,v)$ is an edge with positive flow. In the former case, moving excess from $v$ to $w$ increases the flow on edge $(v,w)$; in the latter case, it decreases the flow on $(w,v)$. We call a pair $(v,w)$ a *residual edge* if $r_f(v,w) > 0$; the *residual graph* $G_f = (V, E_f)$ for a preflow $f$ is the graph whose vertex set is $V$ and whose edge set $E_f$ is the set of residual edges.

The second issue is how to estimate the distance from a vertex to $s$ or to $t$. For this purpose we define a *valid labeling* $d$ to be a function from the vertices to the nonnegative integers and infinity [1], such that $d(s) = n$, $d(t) = 0$, and $d(v) \leq d(w)+1$ for every residual edge $(v,w)$. The intent is that if $d(v) < n$, then $d(v)$ is a lower bound on the actual distance from $v$ to $t$ in the residual graph $G_f$, and if $d(v) \geq n$, then $d(v) - n$ is a lower bound on the actual distance to $s$ in the residual graph. It can be proved by induction that in the latter case $t$ is not reachable from $v$ in $G_f$.)

To describe the algorithm, we also need the following definition. We call a vertex $v$ *active* if $v \in V - \{s,t\}$, $d(v) < \infty$, and $e(v) > 0$.

The maximum flow algorithm begins with the preflow $f$ that is equal to the edge capacity on each edge leaving the source and zero on all other edges, and with some initial sink labeling $d$. The algorithm then repetitively performs, in any order, the *basic operations*, *push* and *relabel*, described in Figure 1.1. When there are no active vertices, the algorithm terminates. A summary of the algorithm appears in Figure 1.2.

[1] According to our definition, distance labels are allowed to be infinite. We shall show, however, that the labels stay finite throughout the execution of the algorithm. Infinite labels are introduced only to simplify the exposition.

*Push*$(v, w)$.
**Applicability:** $v$ is active, $r_f(v, w) > 0$ **and** $d(v) = d(w) + 1$.
**Action:** Send $\delta = \min(e(v), r_f(v, w))$ units of flow from $v$ to $w$ as follows:
$f(v, w) \leftarrow f(v, w) + \delta$; $f(w, v) \leftarrow f(w, v) - \delta$;
$e(v) \leftarrow e(v) - \delta$; $e(w) \leftarrow e(w) + \delta$.

*Relabel*$(v)$.
**Applicability:** $v$ is active **and** $\forall w \in V$, $r_f(v, w) > 0 \Rightarrow d(v) \leq d(w)$.
**Action:** $d(v) \leftarrow \min_{(v,w) \in E_f}(d(w) + 1)$.
(If this minimum is over an empty set, $d(v) \leftarrow \infty$.)

Figure 1.1: Push and relabel operations.

The basic operations modify the preflow $f$ and the labeling $d$. A ***push*** from $v$ to $w$ increases $f(v, w)$ and $e(w)$ by $\delta = \min(e(v), r_f(v, w))$, and decreases $f(w, v)$ and $e(v)$ by the same amount. The push is ***saturating*** if $r_f(v, w) = 0$ after the push and ***nonsaturating*** otherwise. A ***relabeling*** of $v$ sets the label of $v$ to the largest value allowed by the valid labeling constraints.

**Lemma 1.2.1** *If $f$ is a preflow, $d$ is any valid labeling for $f$, and $v$ is any active vertex, then either a push or a relabel operation is applicable to $v$.*

*Proof*: For any residual edge $(v, w)$, the definition of a valid labeling implies that $d(v) \leq d(w)+1$. If a push is not applicable to $v$, then $d(v) < d(w)+1$ for all residual edges $(v, w)$. By the integrality of valid labelings, $d(v) \leq d(w)$ for all residual edges $(v, w)$, and a relabeling is applicable to $v$. ∎

There is one part of the algorithm we have not yet specified: the choice of an initial labeling $d$. The simplest choice is $d(s) = n$ and $d(v) = 0$ for $v \in V - \{s\}$. A more accurate choice (indeed, the most accurate possible choice) is $d(v) = \min(d_{G_f}(v, t), d_{G_f}(v, s) + n)$ for $v \in V$, where $f$ is the initial preflow. The latter labeling can be computed in $O(m)$ time using backward breadth-first searches from the sink and from the source in the residual graph. The resource bounds we shall

```
Procedure Max-Flow (V, E, s, t, c);

    ⟨⟨ initialization ⟩⟩
      ⟨⟨ initialize preflow ⟩⟩
        ∀(v, w) ∈ (V − {s}) × (V − {s}) do begin
                    f(v, w) ← 0; f(w, v) ← 0;
        end;
        ∀w ∈ V do f(s, w) ← u(s, w);
      ⟨⟨ initialize labeling ⟩⟩
        d(s) ← n;
        ∀v ∈ V − {s} do d(v) ← 0;
    ⟨⟨ loop ⟩⟩
      while ∃ a basic operation that applies do
            select a basic operation and apply it;
    return(f);

end.
```

Figure 1.2: The generic maximum flow algorithm. The running time of the algorithm depends on the order in which basic operations are applied and on details of the implementation.

derive for the algorithm are correct for any valid initial labeling. To simplify the proofs, we assume that the algorithm starts with the simple labeling.

## 1.3 Correctness and Termination

We shall prove that the generic algorithm is correct assuming that it terminates and then prove termination.

**Lemma 1.3.1** *The algorithm maintains the invariant that $d$ is a valid labeling.*

*Proof*: We use induction on the number of pushing and relabeling steps. The simple labeling used initially is valid because labels of all vertices other than $s$ are zero, and all edges leaving $s$ are saturated. Given that $d$ is a valid labeling, a relabeling step changing $d(v)$ must produce a new valid labeling. Consider a pushing step

that sends flow from $v$ to $w$. This step may add $(w, v)$ to $G_f$ and may delete $(v, w)$ from $G_f$. Since $d(w) = d(v) - 1$, the addition of $(w, v)$ to $G_f$ does not affect the invariant that $d$ is a valid labeling. The deletion of $(v, w)$ removes the corresponding constraint, which also leaves the labeling valid. ∎

To prove correctness, we use the concept of an augmenting path. An *augmenting path* is a simple path from $s$ to $t$ in the residual graph $G_f$. Our proof of correctness is based on the classical theorem of Ford and Fulkerson [20]:

**Theorem 1.3.2** *A flow $f$ is maximum if and only if there is no augmenting path, i.e. $t$ is not reachable from $s$ in $G_f$.*

**Lemma 1.3.3** *If $f$ is a preflow and $d$ is any valid labeling for $f$, then the sink $t$ is not reachable from the source $s$ in the residual graph $G_f$.*

*Proof*: Assume by way of contradiction that there is an augmenting path $s = v_0, v_1 \ldots, v_l = t$. Then $l < n$ and $(v_i, v_{i+1}) \in E_f$ for $0 \leq i < l$. Since $d$ is a valid labeling, we have $d(v_i) \leq d(v_{i+1}) + 1$ for $0 \leq i < l$. Therefore, we have $d(s) \leq d(t) + l < n$, since $d(t) = 0$, which contradicts $d(s) = n$. ∎

**Theorem 1.3.4** *Suppose that the algorithm terminates and all distance labels are finite at termination. Then the preflow $f$ is a maximum flow.*

*Proof*: If the algorithm terminates and all distance labels are finite, all vertices in $V - \{s, t\}$ must have zero excess, because there are no active vertices. Therefore $f$ must be a flow. This flow is maximum by Lemma 1.3.3 and Theorem 1.3.2. ∎

Now we show that the algorithm terminates and that the distance labels stay finite during the execution of the algorithm. First we prove the following lemma:

**Lemma 1.3.5** *If $f$ is a preflow and $v$ is a vertex with positive excess, then the source $s$ is reachable from $v$ in the residual graph $G_f$.*

*Proof*: Let $S$ be the set of vertices reachable from $v$ in $G_f$, and suppose $s \notin S$. Let $\overline{S} = V - S$. The choice of $S$ implies that for every vertex pair $i, w$ with $i \in S$ and $w \in \overline{S}$, we have $f(w,i) \leq 0$. Thus

$$\begin{aligned} \sum_{i \in S} e(i) &= \sum_{w \in V, i \in S} f(w,i) \\ &= \sum_{w \in \overline{S}, i \in S} f(w,i) + \sum_{w, i \in S} f(w,i) \\ &= \sum_{w \in \overline{S}, i \in S} f(w,i) \\ &\leq 0. \end{aligned}$$

The term $\sum_{i,w \in S} f(w,i)$ in the third line equals zero by antisymmetry. Since $f$ is a preflow, we have $e(i) = 0$ for all $i \in S$, and in particular, we have $e(v) = 0$. ∎

**Lemma 1.3.6** *For any vertex $v$, the distance label $d(v)$ never decreases. An application of a relabeling operation to $v$ increases $d(v)$.*

*Proof*: Since the labeling $d$ is changed using relabeling operations only, it is enough to prove the second statement of the lemma. Suppose a relabeling operation is applicable to $v$. Then for all $w$ such that $(v,w) \in E_f$, we have $d(w) \geq d(v)$, which implies that $\min_{(v,w) \in E_f}(d(w) + 1) > d(v)$, so the relabeling must increase $d(v)$. ∎

We have shown that an application of the relabeling operation to a vertex increases the vertex label. The next lemma shows that the labels cannot increase too much. In particular, the lemma implies that the labels stay finite during an execution of the algorithm.

**Lemma 1.3.7** *At any time during the execution of the algorithm and for any vertex $v \in V$, $d(v) \leq 2n - 1$.*

*Proof*: The lemma is trivial for $v = s$ and $v = t$. Suppose $v \in V - \{s,t\}$. Since the algorithm changes only labels of active vertices, it is enough to prove the lemma for an active vertex $v$. If $v$ is active, then $e(v) > 0$, so by Lemma 1.3.5 there is a simple path from $v$ to $s$ in $G_f$. Let $v = v_0, v_1, \ldots, v_l = s$ be such a path. The length $l$ of the path is at most $n - 1$. Since $d$ is a valid labeling and $(v_i, v_{i+1}) \in E_f$, we have

$d(v_i) \leq d(v_{i+1}) + 1$. Therefore, we have $d(v) = d(v_0) \leq d(v_l) + l \leq d(s) + (n-1) = 2n - 1$. ∎

Lemma 1.3.7 allows us to amortize the work done by the algorithm over increases in vertex labels. The next two lemmas bound the number of relabelings and the number of saturating pushes.

**Lemma 1.3.8** *The number of relabeling operations is at most* $2n-1$ *per vertex and at most* $(2n-1)(n-2) < 2n^2$ *overall.*

*Proof*: Relabeling operations apply only to vertices $v \in V - \{s,t\}$. A relabeling of $v$ increases $d(v)$. The label $d(v)$ is zero initially, and the label can grow to at most $2n-1$. Therefore there are at most $2n-1$ relabelings of each vertex in $V - \{s,t\}$, and the total number of relabelings in at most $(2n-1)(n-2)$. ∎

**Lemma 1.3.9** *The number of saturating push operations is at most* $2nm$.

*Proof*: For any pair of vertices $v$ and $w$, consider the saturating pushes from $v$ to $w$ and from $w$ to $v$. If there are any such pushes, it must be the case that $(v,w) \in E$ or $(w,v) \in E$. Consider a saturating push from $v$ to $w$. In order to push flow from $v$ to $w$ again, the algorithm must first push flow from $w$ to $v$, which cannot happen until $d(w)$ increases by at least two. Similarly, $d(v)$ must increase by at least two between saturating pushes from $w$ to $v$. Since $d(v) + d(w) \geq 1$ when the first push between $v$ and $w$ occurs and $d(v) + d(w) \leq 4n - 3$ when the last such push occurs (by Lemma 1.3.7), the total number of saturating pushes between $v$ and $w$ is at most $2n-1$. Thus the total number of saturating pushes is at most $2n-1$ per edge, for a total over all edges of at most $(2n-1)m < 2nm$. ∎

Next we bound the number of nonsaturating pushing operations.

**Lemma 1.3.10** *The number of nonsaturating pushing operations is at most* $4n^2m$.

*Proof*: Let $\Phi = \sum_{\{v|v \text{ is active}\}}(d(v))$. Each nonsaturating push from $v$ to $w$ causes $\Phi$ to decrease by at least one, since the push makes $v$ inactive and $d(w) = d(v) - 1$. A saturating pushing operation causes $\Phi$ to increase by at most $2n - 1$. The total increase in $\Phi$ due to saturating pushes is at most $(2n - 1) \times 2nm$ by Lemma 1.3.9. The total increase in $\Phi$ over the entire algorithm due to relabeling operations is at most $(2n-1)(n-2)$ by Lemma 1.3.8. Immediately after the initialization $\Phi$ is zero, $\Phi$ is always nonnegative, and at the end of the algorithm $\Phi$ is zero. Thus the total decrease in $\Phi$, and hence the total number of nonsaturating pushing operations, is the total increase, which is at most $(2n-1)2nm + (2n-1)(n-2) \leq 4n^2m$ for $n \geq 4$ (recall the assumption $m \geq n - 1$). ∎

**Theorem 1.3.11** *The generic algorithm terminates after $O(n^2m)$ basic operations.*

*Proof*: Immediate from Lemmas 1.3.8, 1.3.9, and 1.3.10. ∎

The running time of the generic algorithm depends upon the order in which basic operations are applied and the details of implementation, but it is clear that any reasonable sequential implementation of the algorithm will run in polynomial time. In the next section we discuss one possible implementation with an $O(n^2m)$ time bound. We also show that a particular ordering of the basic operations yields an $O(n^3)$ time bound.

We conclude this section with a discussion of a variant of the generic maximum flow algorithm. Let us recall a classical concept from network flow theory, that of a cut. A *cut* $S, \overline{S}$ is a partition of the vertex set $V$ (that is, $S \bigcup \overline{S} = V$ and $S \bigcap \overline{S} = \emptyset$) such that $s \in S$ and $t \in \overline{S}$. The *capacity* of the cut is

$$u(S, \overline{S}) = \sum_{v \in S, w \in \overline{S}} u(v, w).$$

A cut is *minimum* if it has minimum possible capacity. The *max-flow, min-cut* theorem of Ford and Fulkerson [20,19] states that the value of a maximum flow equals the capacity of a minimum cut.

In many applications in which the maximum flow problem occurs, only the maximum flow value or a minimum cut is needed, not an actual maximum flow [61]. For such applications our maximum flow algorithm can be modified to compute a minimum cut and the maximum flow value without actually computing a maximum flow. What we have to say about the maximum flow algorithm in the remainder of the chapter applies to this cut algorithm as well. The only change necessary is to redefine an active vertex to be a vertex $v \in V - \{s,t\}$ such that $e(v) > 0$ and $d(v) < n$. When the modified algorithm terminates, the excess $e(t)$ at the sink is the value of a maximum flow, and the cut $S, \overline{S}$ such that $\overline{S}$ contains exactly those vertices from which $t$ is reachable in $G_f$ is a minimum cut [37]. For this variant of the algorithm, the bounds in Lemmas 1.3.8–1.3.10 can be improved by roughly a factor of two.

## 1.4 Sequential Implementation

Any reasonable implementation of the generic maximum flow algorithm runs in polynomial time. Some implementations, however, are more efficient then others. We shall start with a simple implementation and then refine it to improve efficiency.

As a first step toward obtaining an efficient sequential implementation, we shall describe a simple refinement of the generic algorithm that runs in $O(n^2m)$ time, matching Dinic's bound. We need some data structures to represent the network and the preflow. We call an unordered pair $\{v,w\}$ such that $(v,w) \in E$ or $(w,v) \in E$ an ***undirected edge*** of $G$. We associate the three values $u(v,w)$, $u(w,v)$, and $f(v,w)(= -f(w,v))$ with each undirected edge $\{v,w\}$. Each vertex $v$ has a list of the incident undirected edges $\{v,w\}$, in fixed but arbitrary order. Thus each edge $\{v,w\}$ appears in exactly two lists, the one for $v$ and the one for $w$. Each vertex $v$ has a ***current edge*** $\{v,w\}$, which is the current candidate for a pushing operation from $v$. Initially, the current edge of $v$ is the first edge on the edge list of $v$. The refined algorithm repeats the ***push/relabel*** operation, described in Figure 1.3, until there are no active vertices. We shall discuss the maintenance of active vertices later.

*Push/Relabel*$(v)$.
Applicability: $v$ is active.
Action: Let $\{v, w\}$ be the current edge of $v$.
**If** *push*$(v, w)$ is applicable, apply it;
**else**
**if** $\{v, w\}$ is not the last edge on the edge list of $v$ **then**
replace $\{v, w\}$ as the current edge of $v$ by the next edge on the edge list of $v$;
**else begin**
make the first edge on the edge list of $v$ the current edge;
apply *relabel*$(v)$;
**end.**

Figure 1.3: The push/relabel operation.

The *push/relabel* operation combines the basic operations using the above data structures. When applied to an active vertex $v$, the operation tries to push excess along the current edge $(v, w)$, or advance the current edge if the pushing operation does not apply to the current edge. Advancing the current edge is impossible if this edge is the last edge on the edge list of $v$. In this case, *push/relabel* makes the first edge on the edge list of $v$ the current edge and apply the relabeling operation to $v$. We need to show that push/relabel uses the relabeling operation correctly.

**Lemma 1.4.1** *The push/relabel operation does a relabeling only when the relabeling operation is applicable.*

*Proof*: *Push/relabel* applies the relabeling operation at a vertex $v$ only when $v$ is active. Just before the relabeling, for each edge $(v, w)$, either $d(v) \leq d(w)$ or $r_f(v, w) = 0$, because the distance label $d(v)$ has not changed since $(v, w)$ was the current edge, $d(w)$ never decreases, and $d(w)$ $r_f(v, w)$ cannot increase unless $d(w) > d(v)$. The lemma follows from the definition of a relabeling operation. ∎

The refined algorithm needs one additional data structure, a set $Q$ containing all active vertices. Initially $Q = \{w \in V - \{s, t\} | u(s, w) > 0\}$. Maintaining $Q$ takes only $O(1)$ time per push/relabeling operation. (Such an operation applied to an

*Discharge.*
Applicability: $Q \neq \emptyset$.
Action: Remove the vertex $v$ on the front of $Q$.
(Vertex $v$ must be active.)
**Repeat**
apply *push/relabel*($v$);
**if** $w$ becomes active during this *push/relabel* operation **then**
add $w$ to the rear of $Q$;
**until** $e(v) = 0$ **or** $d(v)$ increases.
**If** $v$ is still active **then** add $v$ to the rear of $Q$.

Figure 1.4: The discharge operation.

edge $\{v, w\}$ may require adding $w$ to $Q$ and/or deleting $v$.)

**Theorem 1.4.2** *The refined algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating pushing step, for a total of $O(n^2m)$ time.*

*Proof*: Let $v$ be a vertex in $V - \{s, t\}$, and let $\Delta_v$ be the number of edges on the edge list of $v$. Relabeling $v$ requires a single scan of the edge list of $v$. By Lemma 1.3.8, the total number of passes through the edge list of $v$ is at most $4n - 1$, one for each of the at most $(2n - 1)$ relabelings of $v$, one before each relabeling as the current edge runs through the list, and one after the last relabeling. Every *push/relabel* operation selecting $v$ either causes a push, changes the current edge of $v$, or increases $d(v)$. The total time spent in *push/relabel* operations selecting $v$ is $O(n\Delta_v)$ plus $O(1)$ time per push out of $v$. Summing over all vertices and applying Lemmas 1.3.9 and 1.3.10 gives the theorem. ∎

To obtain a better running time we need to reduce the number of nonsaturating pushes. We do this in a way similar to that used by Shiloach and Vishkin [66]. Namely, we exploit the freedom we have in selecting vertices for push/relabel operations by using a first-in, first-out selection strategy; i.e., we maintain $Q$ as a queue. The *first-in, first-out algorithm* consists of applying the *discharge* operation until $Q$ is empty. The discharge operation consists of applying *push/relabel* operations

to an active vertex at least until the excess becomes zero or the label of the vertex increases.

There is still some flexibility in this algorithm, namely in how long we keep applying *push/relabel* operations to a vertex $v$. Figure 1.4 describes one extreme case, when we stop as soon as $e(v) = 0$ or $v$ is relabeled. At the other extreme we can continue until $v$ becomes inactive, which may involve several relabelings of $v$. In the sequential case, our analysis is valid for both extremes and all intermediate variants. In the parallel case, our analysis applies only to the version described in Figure 1.4, but can be easily modified to handle the other cases as well.

To analyze the first-in, first-out algorithm, we need to introduce the concept of *passes* over the queue. Pass one consists of the discharging operations applied to the vertices added to the queue during the initialization. Given that pass $i$ is defined, pass $i + 1$ consists of the discharging operations applied to vertices on the queue that were added during pass $i$.

**Lemma 1.4.3** *The number of passes over the queue is at most* $4n^2$.

*Proof*: Define the potential function $\Phi$ to be $\Phi = \max\{d(v) | v \text{ is active}\}$. Consider the effect on $\Phi$ of a single pass over the queue. If no distance label changes during the pass, each vertex succeeds in moving its excess to lower labeled vertices, so $\Phi$ decreases during the pass. If $\Phi$ is not changed by the pass, some vertex label must increase by at least one. If $\Phi$ increases, some vertex label must increase by at least as much as $\Phi$ increases. The total number of passes in which $\Phi$ stays the same or increases is thus at most $2n^2$ by Lemma 1.3.7. Since $\Phi = 0$ initially and $\Phi$ is always nonnegative, the total number of passes in which $\Phi$ decreases is also at most $2n^2$. Hence the total number of passes is at most $4n^2$. ∎

**Corollary 1.4.4** *The number of nonsaturating pushes during the first-in, first-out algorithm is at most* $4n^3$.

*Proof*: There is at most one nonsaturating push per vertex in $V - \{s, t\}$ per pass. ∎

**Theorem 1.4.5** *The first-in, first-out algorithm runs in $O(n^3)$ time.*

*Proof*: Immediate from Theorem 1.4.2 and Corollary 1.4.4. ∎

An alternative strategy for vertex selection, which we call the *maximum distance method*, is to always select a vertex $v$ in $Q$ with $d(v)$ maximum. This strategy also gives an $O(n^3)$ running time, as a proof similar to that of Theorem 1.4.5 shows.

## 1.5 Use of Dynamic Trees

We have now matched the $O(n^3)$ time bound of Karzanov's algorithm. To obtain a better bound, we must reduce the time per nonsaturating pushing operation below $O(1)$. We do so by using the dynamic tree data structure of Sleator and Tarjan [68,69,71]. This data structure allows us to maintain a set of vertex-disjoint rooted trees in which each vertex $v$ has an associated real value $g(v)$, possibly $\infty$ or $-\infty$. We regard a tree edge as directed toward the root, i.e. from child to parent. We denote the parent of a vertex $v$ by $p(v)$. We adopt the convention that every vertex is both an ancestor and a descendant of itself. The tree operations we shall need are described in Figure 1.5.

The total time for a sequence of $l$ tree operations starting with a collection of single-vertex trees is $O(l \log k)$, where $k$ is an upper bound on the maximum number of vertices in a tree. (The implementation of dynamic trees presented in [69,71] does not support *find-size* operations, but it is easily modified to do so. See [37].)

In our application the edges of the dynamic trees are a subset of the current edges of the vertices. The current edge $\{v, w\}$ of a vertex $v \in V - \{s, t\}$ is *eligible* to be a dynamic tree edge (with $p(v) = w$) if $d(v) = d(w) + 1$ and $r_f(v, w) > 0$. Not all eligible edges are tree edges, however. The value $g(v)$ of a vertex $v$ in its dynamic tree is $r_f(v, p(v))$ if $v$ has a parent and $\infty$ if $v$ is a tree root. Initially, each vertex is in a one-vertex dynamic tree and has value $\infty$. We limit the maximum tree size

*find-root*($v$): Find and return the root of the tree containing vertex $v$.

*find-size*($v$): Find and return the number of vertices in the tree containing vertex $v$.

*find-value*($v$): Compute and return $g(v)$.

*find-min*($v$): Find and return the ancestor $w$ of $v$ of minimum value $g(w)$. In case of a tie, choose the vertex $w$ closest to the root.

*change-value*($v, x$): Add real number $x$ to $g(w)$ for all ancestors $w$ of $v$. (We adopt the convention that $\infty + (-\infty) = 0$.)

*link*($v, w$): Combine the trees containing vertices $v$ and $w$ by making $w$ the parent of $v$. This operation does nothing if $v$ and $w$ are in the same tree or if $v$ is not a tree root.

*cut*($v$): Break the tree containing $v$ into two trees by deleting the edge from $v$ to its parent. This operation does nothing if $v$ is a tree root.

Figure 1.5: Dynamic tree operations.

to $k$, where $k$ is a parameter to be chosen later.

By using appropriate tree operations we can push flow along an entire path in a tree, either causing a saturating push or moving flow excess from some vertex in the tree all the way to the tree root. By combining this idea with a careful analysis, we are able to show that the number of times a vertex is added to $Q$ is $O(nm + n^3/k)$. At a cost of $O(\log k)$ for each tree operation, the total running time of the algorithm is $O((nm + n^3/k)\log k)$, which is minimized to within a constant factor at $O(nm \log(n^2/m))$ for the choice $k = n^2/m$.

The details of the improved algorithm, which we call the *dynamic tree algorithm*, are as follows. The heart of the algorithm is the procedure *send*($v$) defined in Figure 1.6, which pushes excess from a nonroot vertex $v$ to the root of its tree, cuts edges saturated by the push, and repeats these steps until $e(v) = 0$ or $v$ is a tree root.

At the top level, the dynamic tree algorithm is exactly the same as the first-in, first-out algorithm of Section 1.4: we maintain a queue $Q$ of active vertices and repeatedly perform discharging operations until $Q$ is empty. However, we replace the push/relabel operation with the *tree-push/relabel operation* described in Figure

*Send*($v$).
**Applicability:** $v$ is active.
**Action:** **While** *find-root*($v$) $\neq v$ **and** $e(v) > 0$ **do begin**
send $\delta \leftarrow \min(e(v), \textit{find-value}(\textit{find-min}(v)))$ units of flow
along the tree path from $v$ by performing *change-value*($v, -\delta$);
**while** *find-value*(*find-min*($v$)) $= 0$ **do begin**
$i \leftarrow$ *find-min*($v$);
perform *cut*($i$) followed by *change-value*($i, \infty$);
**end;**
**end.**

Figure 1.6: The Send operation.

1.7.

A *tree-push/relabel* operation applies to an active vertex $v$ that is the root of a dynamic tree. There are two main cases. The first case occurs if the current edge $\{v, w\}$ of $v$ is eligible for a pushing operation. If the trees containing $v$ and $w$ together have at most $k$ vertices, we link these trees by making $w$ the parent of $v$ and do a send operation from $v$. If these trees together contain more then $k$ vertices, we do an ordinary pushing operation from $v$ to $w$ followed by a send from $w$. The second case occurs if the edge $\{v, w\}$ is not eligible for a pushing operation. In this case we update the current edge of $v$ and relabel $v$ if necessary. If $v$ is relabeled, we cut all tree edges entering $v$, to maintain the invariant that all dynamic tree edges are eligible for pushing operations.

It is important to realize that this algorithm stores values of the preflow $f$ in two different ways. If $\{v, w\}$ is an edge that is not a dynamic tree edge, $f(v, w)$ is stored explicitly, with $\{v, w\}$. If $\{v, w\}$ is a dynamic tree edge, with $w$ the parent of $v$, then $g(v) = u(v, w) - f(v, w)$ is stored implicitly in the dynamic tree data structure. Whenever a tree edge $(v, w)$ is cut, $g(v)$ must be computed and $f(v, w)$ updated to its current value. In addition, when the algorithm terminates, preflow values must be computed for all edges remaining in dynamic trees.

Two observations imply that the dynamic tree algorithm is correct. First, any

*Tree-Push/Relabel*$(v)$.
*Applicability:* $v$ is an active tree root.
*Action:* Let $\{v, w\}$ be the current edge of $v$.
**(1) If** $d(v) = d(w) - 1$ **and** $r_f(v, w) > 0$ **then begin**
**(1a) If** *find-size*$(v)$ + *find-size*$(w) \leq k$ **then begin**
make $w$ the parent of $v$ by performing
*change-value*$(v, -\infty)$, *change-value*$(v, r_f(v, w))$, **and** *link*$(v, w)$;
push excess from $v$ to $w$ by performing *send*$(v)$;
**end;**
**(1b) else** $\langle\langle$ *find-size*$(v)$ + *find-size*$(w) > k\rangle\rangle$ **begin**
apply a pushing operation to move excess from $v$ to $w$;
perform *send*$(w)$;
**end;**
**(2) else** $\langle\langle d(v) > d(w) - 1$ **or** $r_f(v, w) = 0\rangle\rangle$
**(2a) if** $\{v, w\}$ is not the last edge on the edge list of $v$ **then**
replace $\{v, w\}$ as the current edge by the next edge on the list;
**(2b) else** $\langle\langle\{v, w\}$ is the last edge on the edge list of $v\rangle\rangle$ **begin**
make the first edge on the list the current one;
perform *cut*$(i)$ and *change-value*$(i)$ for every child $i$ of $v$;
apply a relabeling operation to $v$.
**end.**

Figure 1.7: The tree-push/relabel operation.

edge $\{v, w\}$ that is in a dynamic tree has $d(v) = d(w) + 1$. Therefore in case (1a) of *tree-push/relabel*, vertices $v$ and $w$ are in different trees, and the algorithm never attempts to link a dynamic tree to itself. Second, a vertex $v$ that is not a tree root can have positive excess only in the middle of case (1) of a *tree-push/relabel* operation. To see this, note that only in this case does the algorithm add excess to a nonroot vertex, and this addition of excess is followed by a *send* operation that moves the nonroot excess to one or more roots.

**Lemma 1.5.1** *The dynamic tree algorithm runs in* $O(nm \log k)$ *time plus* $O(\log k)$ *time per addition of a vertex to* $Q$.

*Proof*: The condition in subcase (1a) of *tree-push/relabel* guarantees that the maximum size of any dynamic tree is $k$. Thus the time per dynamic tree operation is

$O(\log k)$. Each *tree-push/relabel* operation takes $O(1)$ time plus $O(1)$ tree operations plus $O(1)$ tree operations per cut operation (in invocations of *send* and in subcase (2b)) plus time for relabeling (in subcase (2b)). The total relabeling time is $O(nm)$. The total number of cut operations is at most the number of *link* operations, which is at most $4(n-1)m$ by a proof like that of Lemma 1.3.9. (Another way of getting a bound on the number of cut and link operations is to observe that a cut operation corresponds to a saturating push or to an edge scan during relabeling, and the number of link operations exceeds the number of cut operations by at most $n-1$.) The total number of *tree-push/relabel* operations is $O(nm)$ plus one per addition of a vertex to $Q$. Combining these observations gives the lemma. ∎

We define passes over the queue $Q$ exactly as in Section 1.4. The proof of Lemma 1.4.3 remains valid, which means that the number of passes is at most $4(n-1)^2$.

The next lemma is the crucial part of the analysis.

**Lemma 1.5.2** *The number of times a vertex is added to $Q$ is $O(nm + n^3/k)$.*

*Proof*: A vertex $v$ can be added to $Q$ only after $d(v)$ increases, which happens at most $2(n-1)^2$ times, or as a result of $e(v)$ increasing from zero, which can happen only in subcases (1a) and (1b) of *tree-push/relabel*. In either subcase, the number of vertices added to $Q$ is at most one more than the number of cuts performed during the invocation of *send* in the subcase. Thus the number of additions to $Q$ in subcases (1a) and (1b) is at most $2nm$ (the maximum number of cuts) plus the number of occurrences of the subcases. There are at most $2nm$ occurrences of (1b) in which the invocation of *send*$(w)$ causes a cut, and at most $2nm$ occurrences of (1b) in which the push from $v$ to $w$ is saturating. Let us call an occurrence of (1b) *nonsaturating* if it adds a vertex to $Q$ but causes neither a cut nor a saturating push. It remains for us to count the number of nonsaturating occurrences.

We need a few definitions. For any vertex $u$, we denote the dynamic tree containing $u$ by $T_u$ and the number of vertices it contains by $|T_u|$. Tree $T_u$ is *small* if $|T_u| \leq k/2$ and *large* otherwise. At any time, and in particular at the beginning of any pass, there are at most $2n/k$ large trees.

Consider a nonsaturating occurrence of (1b) during a given pass, say pass $i$. The condition in (1b) guarantees that either $T_v$ or $T_w$ is large, giving us two cases to consider.

Suppose $T_v$ is large. Vertex $v$ is the root of $T_v$. The nonsaturating occurrence of (1b) removes all the excess from $v$, which means that a nonsaturating occurrence can apply to a given vertex $v$ only once during a given pass. If $T_v$ has changed since the beginning of pass $i$, we charge the occurrence of (1b) to the link or cut that changed $T_v$ most recently before the occurrence. The number of such occurrences over all passes is at most one per link and two per cut, for a total of at most $6nm$. (A link forms one new tree; a cut, two.) If $T_v$ has not changed since the beginning of pass $i$, we charge the occurrence of (1b) to $T_v$. Since $T_v$ is large and there are at most $2n/k$ large trees at the beginning of pass $i$, there are at most $2n/k$ such charges per pass, for a total of at most $4n^3/k$ over all passes.

Suppose on the other hand that $T_w$ is large. The occurrence of (1b) adds the root of $T_w$, say $r$, to $Q$ (otherwise this occurrence of (1b) need not be counted). A given vertex $r$ can be added to $Q$ at most once during a given pass. If $T_w$ has changed since the beginning of pass $i$, we charge the occurrence of (1b) to the link or cut that changed $T_w$ most recently before the occurrence. The number of such occurrences over all passes is at most $6nm$. If $T_w$ has not changed, we charge the occurrence to $T_w$. The number of such charges over all passes is at most $4n^3/k$.

Summing our estimates, we find that there are at most $2n^2 + 20nm + 8n^3/k$ additions to $Q$ altogether, giving the lemma. ∎

**Theorem 1.5.3** *The dynamic tree algorithm runs in $O(nm\log(n^2/m))$ time if $k$ is chosen equal to $n^2/m$.*

*Proof*: Immediate from Lemmas 1.5.1 and 1.5.2. ∎

As in Section 1.4, we can replace first-in, first-out selection of vertices for discharging steps by maximum distance selection, and still obtain the same running time bound.

**Procedure** *Pulse*.
For all active vertices $v$ in parallel **do begin**
⟨⟨ **stage 1**⟩⟩
push flow from $v$ **until** $e(v) = 0$ **or** $\forall w$ such that $d(w) = d(v) - 1$, $r_f(v, w) = 0$.
⟨⟨ **stage 2**⟩⟩
**If** $e(v) > 0$ **then** $d'(v) \leftarrow \min_{w|r_f(v,w)>0}(d(w) + 1)$ **then begin**
$d(v) \leftarrow d'(v)$;
broadcast $d(v)$ to all neighbors of $v$;
**end.**
⟨⟨ **stage 3**⟩⟩
Add flow pushed to $v$ in stage 1 to $e(v)$.
**end.**

Figure 1.8: The Pulse operation.

## 1.6 Parallel and Distributed Implementation

The synchronous parallel version of our algorithm is a modification of the first-in, first-out algorithm of Section 1.4. The algorithm proceeds in pulses, each of which consists of a number of operations applied in parallel. Each pulse is divided into three stages. The pushing of flow is done during the first stage, the relabeling of vertices is done in the second stage, and flow pushed to a vertex in the first stage is added to its excess in the third stage. We make three changes in the algorithm. First, we restrict the algorithm so that it stops processing a vertex $v$ as soon as $e(v) = 0$ or $v$ is relabeled. Second, instead of using a queue for selection of vertices to be processed, we process all active vertices in parallel. Third, the flow pushed to a vertex $v$ during a parallel step is not added to $e(v)$ until the third stage. To be more precise, the parallel version consists of repeating the *pulse* step described in Figure 1.8 until there are no active vertices.

The parallel algorithm is almost a special case of the first-in, first-out algorithm, the only difference being in the values used in relabeling and flow excess computations: in the first-in, first-out algorithm, these computations in pass $i$ use the most recent label and excess values, some of which may have been computed earlier

in pass $i$. Nevertheless, a proof just like that of Lemma 1.4.3 gives the following analogous result for the parallel algorithm:

**Lemma 1.6.1** *The number of pulses made by the parallel algorithm is at most* $4n^2$.

**Corollary 1.6.2** *The number of nonsaturating pushes made by the parallel algorithm is at most* $4n^3$.

For the distributed implementation of this algorithm, our computing model is as follows [31,2]. We allow each vertex $v$ of the graph to have a processor with an amount of memory proportional to $\Delta_v$, the number of neighbors of $v$. This processor can communicate directly with the processors at all neighboring vertices. We assume that local computation is much faster than inter-processor communication. Thus as a measure of computation time we use the number of rounds of message-passing. We are also interested in the total number of messages sent.

A synchronous distributed implementation of the parallel algorithm works as follows. Each vertex processed during a pulse sends updated flow values to the appropriate neighbors. New vertex labels are also transmitted to neighbors, but only at the end of the pulse. Since flow always travels in the direction from larger to smaller labels, this delaying of the label broadcasting until the end of a pulse guarantees that flow only travels through an edge in one direction during a pulse. An easy analysis shows that in the synchronous case the distributed algorithm takes $O(n^2)$ rounds of message-passing and a total of $O(n^3)$ messages.

For parallel implementation, our computing model is a PRAM [21] without concurrent writing. The implementation in this model is very similar to that of the distributed implementation, except that computations on binary trees must be performed to allow each vertex to access its incident edges fast. Because of these binary trees, each pulse takes $O(\log n)$ time, and the parallel time of the algorithm is $O(n^2 \log n)$. The ideas of Shiloach and Vishkin [66] apply to our algorithm to show that $O(n)$ processors suffice to obtain the $O(n^2 \log n)$ time bound. See [66] for details. We describe a different parallel implementation of the algorithm in Chapter 3.

Now we discuss two implementations of the algorithm in the asynchronous distributed model of parallel computation [31,2]. Awerbuch (private communication, 1985) has observed that in the asynchronous case, the synchronization protocol of [2] can be used to implement the algorithm in $O(n^2 \log n)$ rounds and $O(n^3)$ messages. The same bounds can be obtained for the Shiloach-Vishkin algorithm [66], but only by allowing more memory per processor: the processor at a vertex $v$ needs $O(n\Delta_v)$ storage. Vishkin (private communication) has reduced the space required by this algorithm to a total of $O(n^2)$ (from $O(nm)$ ). Nevertheless, our algorithm has an advantage in situations where memory is at a premium.

The algorithm can be modified to work in the asynchronous model without the use of the synchronization protocol, achieving better running time but using more messages. This asynchronous version of the algorithm synchronizes locally using *acknowledgments*. When a vertex $v$ pushes its excess to vertex $w$ such that, according to the local information at $v$, $d(v) = d(w)+1$, it sends a message $(v, \delta, d(v))$ and updates $e(v)$. The vertex $v$ will not push flow to $w$ again or change $d(v)$ until $v$ receives an acknowledgment from $w$. When a vertex $w$ receives a message $(v, \delta, d(v))$, it first checks if $d(v) = d(w) + 1$ (because the value of $d(w)$ in the processor $v$ at the time it sent the message may be out of date). If $d(v) = d(w) + 1$, then $w$ sends to $v$ a message of the form $(\text{accept}, w, \delta, d(w))$. Otherwise it sends to $v$ a message of the form $(\text{reject}, w, \delta, d(w))$, where $d(w)$ is the correct value of the distance label of $w$. The accepting or rejecting messages serve as acknowledgments. In addition, a rejecting message causes $v$ to update its excess, its local value of $d(w)$, and $d(v)$ if necessary. When a distance label of a vertex increases, it informs its neighbors about the new value of the label.

**Theorem 1.6.3** *The asynchronous distributed implementation of the algorithm that uses acknowledgments runs in $O(n^2)$ time using $O(n^2m)$ messages and $O(\Delta_p)$ memory per processor.*

*Proof*: To analyze the message complexity of the algorithm, note that the total number of messages is the number of messages generated by the distance label increases plus twice the number of (accepting or rejecting) acknowledge messages.

The number of messages generated by the distance label increases is at most $(2n - 1)m$. There is at most one rejecting message per edge per distance label increase, for a total of $(2n - 1)m$ (by Lemma 1.3.7). The same arguments as in the proofs of Lemmas 1.3.9 and 1.3.10 give $(2n - 1)m$ and $4n^2m$ bounds on the number of accepting messages corresponding to saturating and nonsaturating pushes. The total message complexity of the algorithm is thus $O(n^2m)$.

To bound the running time of the algorithm, we need to introduce a unit of time. Given an execution of an algorithm, we define a time unit to be the longest time from a point when a message is originated by a sender to the point when the message is processed by the receiver. For example, a push-acknowledgment pair of operations takes two units of time. Note that if during a time interval $(t, t + 4]$ no vertex label increased, then the $\Phi$ function defined as in the proof of Lemma 1.4.3 must decrease during this time interval. To see this, observe that during the time interval $[t + 1, t + 4)$ each vertex has correct information about distance labels of its neighbors, so all pushes initiated during the time interval $[t + 1, t + 2)$ are accepted by time $t + 4$. A proof similar to that of Lemma 1.4.3 yields an $O(n^2)$ bound on the running time of the algorithm. ∎

An alternative way to obtain a fast distributed or parallel algorithm is to use a parallel version of maximum distance selection: during each pulse, apply push/relabel steps to every active vertex $v$ for which $d(v)$ is maximum. This requires a preprocessing step at the beginning of each pulse to compute the maximum $d(v)$, but it simplifies other calculations, since during a given pulse a vertex cannot both send and receive flow, which allows the computations of flow excess to proceed concurrently with the push/relabel steps.

## 1.7 Remarks

Our concluding remarks concern three issues: (i) better bounds, (ii) exact distance labeling, and (iii) efficient practical implementation. Regarding the possibility of obtaining better bounds for the maximum flow problem, it is interesting to note

that the bottleneck in the sequential version of our algorithm is the nonsaturating pushes, whereas the bottleneck in the parallel version is the saturating pushes. Recently, Ahuja and Orlin [1] have devised a scaling algorithm based on the approach described in this chapter. Their algorithm runs in $O(nm + n^2 \log U)$ time (assuming that the edge capacities are integers not exceeding $U$). This improves Gabow's bound of $O(nm \log U)$ mentioned in the introduction. We wonder whether an $O(nm)$ sequential time bound can be obtained through more careful handling of the nonsaturating pushes, possibly avoiding the use of the dynamic tree data structure. Perhaps also an $O(m(\log n)^k)$ parallel time bound can be obtained through the use of a parallel version of the dynamic tree data structure.

It is possible to modify our algorithm so that when a pushing operation is executed, each distance label is exactly the distance to the sink or to the source in the residual graph (i.e. if $d(v) > 0$, then $d(v) = d_{G_f}(v,t)$; if $d(v) \geq n$ then $d(v) = d_{G_f}(v,s) + n$). The modification involves a stronger interpretation of the current edge of a vertex, which should be unsaturated and lead to a vertex with a smaller label. If a pushing step saturates the current edge of $v$, a new current edge is found by scanning the edge list of $v$ and relabeling if the end of the list is reached, as in a push/relabel step. If a relabeling step changes the label of $v$, the current edge must be updated for each vertex $i$ such that $(i,v)$ is the current edge of $i$. One can show that these computations take $O(nm)$ time in total during the algorithm.

Whether maintaining exact distance labels improves the practical performance of the algorithm is not clear, because the work of maintaining the exact labels may exceed the extra work due to nonexact labels. The above observation, however, suggests that as long as we are interested in an $\Omega(nm)$ upper bound on an implementation of the generic algorithm, we can assume that the exact labeling is given to us for free.

Our algorithm is practical. In chapter 3, we describe an implementation of the algorithm and experimental results obtained using the implementation. The algorithm was also used in the context of computational physics [58]. In a prac-

tical implementation it is important to make the algorithm as fast as possible. We offer a heuristic that may speed up the algorithm. The heuristic periodically updates the distance labels by performing breadth-first searches backwards from the sink and source in the residual graph. These searches compute, for each vertex $v$, the distances $d_{G_f}(v,s)$ and $d_{G_f}(v,t)$. The new distance label of $v$ is set to $\min(d_{G_f}(v,s), d_{G_f}(v,t)+n)$. There are several possible strategies for deciding when to recompute labels. One is to do so after every $n$ relabeling operations. Another is to do so every time an edge into the sink is saturated or an edge out of the source has its flow reduced to zero. Neither of these strategies affects the asymptotic time bound of the algorithm, but they may improve its practical performance.

Another important issue in a practical implementation is what strategy to use for selecting vertices for discharging steps. Although the best theoretical bounds we have obtained are for first-in, first-out and maximum distance selection, other strategies, such as last-in, first-out and maximum excess, deserve consideration. Ahuja-Orlin scaling algorithm [1], for example, selects vertices with excess larger then a certain threshold $h$ in such a way that no excess greater then $2h$ is created during a scaling iteration.

# Chapter 2

# The Minimum-Cost Flow Problem

## 2.1 Introduction

In this chapter we introduce a method that allows to apply techniques developed for the maximum flow problem to the more general minimum-cost flow problem. The method incorporates the idea of cost scaling used in the algorithms of Röck [62] and of Bland and Jensen [8], and the concept of relaxed complementary slackness conditions embodied in the algorithms of Bertsekas [7.6] and of Tardos [70]. We show how to use both the maximum flow approach of Dinic [14] and the maximum flow approach of Chapter 1 in the context of this method.

The minimum-cost flow problem is that of finding a feasible flow of minimum cost in a network with capacity constraints and costs on edges. This problem has a wider range of applications then the maximum flow problem discussed in the previous chapter. Extensive discussion of the problem and its applications appear in the books of Ford and Fulkerson [19], Lawler [52], Papadimitriou and Steiglitz [60], and Tarjan [71].

All known polynomial-time algorithms for the problem are based on the idea of *scaling*.[1] This idea was introduced by Edmonds and Karp [15], who used the idea to design the first polynomial-time algorithm for the problem. Scaling algorithms

[1] For applications of scaling to other problems, see [26].

for the minimum-cost flow problem work by solving a sequence $P_0, P_1, \ldots, P_L$ of minimum-cost flow problems. The problem $P_i$ is obtained by considering the $i$ most significant bits of the capacities (capacity scaling) or the costs (cost scaling). The first problem $P_0$ is easy to solve because all the capacities or costs are equal to zero. A solution of the problem $P_i$ helps to solve the next problem $P_{i+1}$. If all of the relevant values have at most $L$ bits, the solution of $P_L$ is the desired solution.

Table 2.1 summarizes polynomial-time algorithms for the minimum-cost flow problem. The running time of the algorithms is given in terms of the number $n$ of vertices, the number $m$ of edges, the maximum absolute value $U$ of capacities, and the maximum absolute value $C$ of costs $C$. When $U$ (or $C$) appears in a bound, the capacities (or the costs) are assumed to be integer. When stating the running times, we assume the best time bounds known for the shortest path and maximum flow subroutines used by some of these algorithms: $O(m + n \log n)$ for the shortest path subroutine [22] and $O(nm \log(n^2/m))$ for the maximum flow subroutine (see Chapter 1). Algorithms 1, 2, 5, 6, and 8 use the capacity scaling and algorithms 3, 4, 7, and 9 use cost scaling. The algorithms 4, 5, 6, and 8 are strongly polynomial: their running time does not depend on $U$ or $C$.

Since the running times of the algorithms in Table 2.1 are expressed in terms of different parameters, the algorithms cannot be compared directly. The previous algorithms 1-8, in the three comparable groups, rank as follows. Algorithms 1 and 2 give the best capacity-dependent bound, algorithms 3 and 7 give the best cost-dependent bound, and algorithm 8 gives the best strongly polynomial bound. For most applications, however, one can assume that $U = n^{O(1)}$ and $C = n^{O(1)}$. Under these assumptions, the bound of $O(m \log(n)(m + n \log n))$ achieved by the capacity-scaling algorithms 1 and 2 is the best among the previous algorithms.

In this chapter we present a general approach to the minimum-cost flow problem. The approach combines methods for solving the maximum flow problem with successive approximation techniques. We use this approach to construct algorithms for the problem with upper bounds of $O(nm \log(n) \log(nC))$, $O(n^{5/3} m^{2/3} \log(nC))$, and $O(n^3 \log(nC))$, which significantly improves the best previous cost-dependent

| # | Date | Discoverer | Running Time | References |
|---|---|---|---|---|
| 1 | 1972 | Edmonds and Karp | $O(m \log(U)(m + n \log n))$ | [15] |
| 2 | 1980 | Röck | $O(m \log(U)(m + n \log n))$ | [62] |
| 3 | 1980 | Röck | $O(n \log(C)(nm \log(n^2/m)))$ | [62] |
| 4 | 1984 | Tardos | $O(m^4)$ | [70] |
| 5 | 1984 | Orlin | $O(m^2 \log(n)(m + n \log n))$ | [59] |
| 6 | 1985 | Fujishige | $O(m^2 \log(n)(m + n \log n))$ | [23] |
| 7 | 1985 | Bland and Jensen | $O(n \log(C)(nm \log(n^2/m)))$ | [8] |
| 8 | 1986 | Galil and Tardos | $O(n^2 \log(n)(m + n \log n))$ | [30] |
| 9 | 1987 | Goldberg and Tarjan | $O(\min(nm \log n, n^{5/3}m^{2/3}, n^3 \log(nC))$ | [38] |

Table 2.1: Polynomial-time algorithms for the minimum-cost flow problem. Algorithm 9 is presented in this chapter.

bound achieved by algorithms 3 and 7 in the table, as well as the best bound under the assumptions $U = n^{O(1)}$ and $C = n^{O(1)}$ discussed above. The algorithm gives the best bound on the complexity of the problem for $C = o(n^n)$ and $C = o(U^n/n)$. Our approach is in many respects similar to the approaches taken by Bland and Jensen [8] to develop algorithm 7, and by Bertsekas [6] to develop an exponential-time algorithm for the problem. Our techniques are more powerful, however, and lead to more efficient algorithms.

The new algorithm works by successive approximation. It starts by finding an approximate solution and then iteratively improves the current solution, each time doubling the quality of approximation by halving the error parameter $\epsilon$. The inner loop subroutine that improves the approximation is based on generalizations of techniques for solving the maximum flow problem. When the error parameter is small enough, the current solution is optimal, and the algorithm terminates. To measure the quality of a solution, we use the notion of $\epsilon$-optimality, which is related to the classical technique of perturbing a linear programming problem to avoid degeneracy (see, for example, [32]). The notion of $\epsilon$-optimality is motivated by the relaxation of the complementary slackness conditions [7,70]. The termination condition used in our algorithm is due to Bertsekas [6].

Our approach can be viewed as a generalization of the cost-scaling approaches of Röck [62] and of Bland and Jensen [8]. The optimal solution to the problem $P_i$ solved by a cost-scaling algorithm is an $\epsilon$-optimal solution to the original problem with $\epsilon = 2^{\lceil \log C \rceil + 1 - i}$, so the error decreases by a factor of two from one scaling iteration to another. In fact, earlier versions of our algorithm used the cost-scaling approach. However, the use of true costs throughout the algorithm simplifies its analysis and implementation.

Our successive approximation approach is different from the traditional scaling approaches in one important way. In traditional scaling, each problem $P_i$ is solved exactly, even though the data used to define the problem $P_i$ is imprecise. Our approach can be interpreted as solving the imprecise subproblems approximately, with the error parameter being within a constant factor of the precision of the problem data. The work saved by solving the intermediate problems approximately is one of the reasons for the improved efficiency of our method. This observation can be used to improve other scaling algorithms. The minimum-cost flow problem is closely related to other flow-like problems and to the linear programming problem, so the techniques developed in this chapter may apply to the other problems as well.

Some of the results of this section are based on observations of Robert Tarjan. These results include Lemma 2.6.5, which simplifies implementations of the generic subroutine and the corresponding analysis, and a way to combine the successive approximation approach with Dinic's layered network approach, which leads to the results described in section 2.9. The results presented in this chapter will also appear in [38].

## 2.2 Definitions and Notation

In this section we define the *minimum-cost circulation problem* and introduce the notation and terminology used throughout the chapter. The minimum-cost circulation problem is a generalization of the maximum flow problem discussed in Chapter 1. The minimum-cost flow problem is also a special case of the linear program-

ming problem and is usually defined in linear programming terms. Although we use several theorems which have their roots in the theory of linear programming, most arguments presented in this chapter are graph theoretic. Consequently, we formulate the problem in graph-theoretic terms. Our formulation of the problem is equivalent to other formulations of the minimum-cost flow and minimum-cost circulation problems that can be found in the books and papers referred in the introduction to this chapter.

A *circulation network* is a directed graph $G = (V, E)$ with upper and lower capacity bounds and costs on edges. Circulation networks are different from flow networks defined in Section 1.2 in three ways: circulation networks have costs on edges, lower bounds on edge capacities, but they do not have sources or sinks. As before, we denote the size of $V$ by $n$ and the size of $E$ by $m$, and we assume that $m \geq n - 1 \geq 4$ (as in Section 1.2). We call an unordered pair $\{v, w\}$ such that $(v, w) \in E$ or $(w, v) \in E$ an *undirected edge* of $G$. For notational convenience, we extend the capacity functions and the cost function to all pairs of vertices. Let $R$ denote the set of real numbers. The capacity bounds are given by functions $u : V \times V \rightarrow R$ and $l : V \times V \rightarrow R$ with the following constraints for all $(v, w) \in V \times V$:

$$l(v, w) \leq u(v, w) \quad \text{(consistency constraints)}, \tag{2.1}$$

$$u(v, w) = -l(w, v) \quad \text{(capacity antisymmetry constraints)}, \tag{2.2}$$

$$l(v, w) = u(v, w) = 0 \text{ if } (v, w) \notin E \text{ and } (w, v) \notin E. \tag{2.3}$$

Property (2.3) is required to extend the capacity functions to all pairs of vertices.

A *pseudoflow*[2] is a function $f : V \times V \rightarrow R$ satisfying the following constraints for all $(v, w) \in V \times V$:

$$l(v, w) \leq f(v, w) \leq u(v, w), \quad \text{(capacity constraints)}, \tag{2.4}$$

[2]The concept of a pseudoflow is different from the preflow concept of Karzanov defined in Section 1.1 in that the flow conservation constraints are completely dropped.

$$f(v,w) = -f(w,v) \quad \text{(flow antisymmetry constraints)}. \tag{2.5}$$

A *circulation* is a pseudoflow that satisfies

$$\sum_{w \in V} f(v,w) = 0 \quad \text{(conservation constraints)} \tag{2.6}$$

for all $v \in V$.

We assume that the costs of edges are given by a cost function $c : V \times V \to R$ that satisfies the following constraints for all $(v,w) \in V \times V$:

$$c(v,w) = -c(w,v) \quad \text{(cost antisymmetry constraints)}. \tag{2.7}$$

We extend the cost function to pairs of vertices by defining $c(v,w) = 0$ for all $(v,w)$ such that $(v,w) \notin E$ and $(w,v) \notin E$. The *cost of a circulation* $f$ is given by the following expression:

$$\frac{1}{2} \sum_{(v,w) \in V \times V} c(v,w) f(v,w). \tag{2.8}$$

(The factor of $1/2$ appears because we count the cost of the flow between each pair of vertices twice.) The *minimum-cost circulation* problem is to find a circulation of minimum cost (an *optimal circulation*).

*Remark*: We refer to equations (2.2), (2.5), and (2.7) as the *antisymmetry constraints*. One should think of a positive and a negative direction for each undirected edge of $G$, with the capacity and cost constraints given for the positive direction and derived for the negative direction using the antisymmetry constraints.

Another important concept related to the problem is the concept of *vertex prices*. To gain intuition, consider a vertex $v$ and a real number $x$. Suppose that we add $x$ to prices of all edges going into $v$ and subtract $x$ from prices of all edges going

out of $v$. Because of the conservation constraints at $v$, the cost of $f$ is not changed by this transformation (the cost of each unit of flow going into $v$ increases by $x$, and the cost of each unit of flow going out of $v$ decreases by $x$). Therefore the transformed problem is equivalent to the original one. To define the prices formally, we introduce a *price function* $p : V \rightarrow R$, and define the price of a vertex $v$ to be $p(v)$. The *reduced cost* function $c_p$ is defined by $c_p(v,w) = c(v,w) - p(v) + p(w)$. In the linear programming interpretation of the problem, the prices correspond to dual variables.

Given a pseudoflow $f$, we define the *residual capacity function* $r_f : V \times V \rightarrow R$ by $r_f(v,w) = u(v,w) - f(v,w)$. The *residual graph* $G_f = (V, E_f)$ is the directed graph with vertex set $V$ containing all edges with positive residual capacity: $E_f = \{(v,w) | r_f(v,w) > 0\}$. The *balance* $b_f(v)$ of a vertex $v$, is the difference between the incoming and outgoing flows, or, more formally, the function $b_f : V \times V \rightarrow R$ defined by $b_f(v) = \sum_{w \in V} f(w,v)$. If $f$ is a circulation, then $b_f(v) = 0$ for all $v$. Given a pseudoflow $f$, we say that a vertex $v$ is *active* if $b_f(v) > 0$. Note that $\sum_{v \in V} b(v) = 0$ for any pseudoflow, so a pseudoflow is a circulation if and only if there are no active vertices.

We also need the following standard definitions. An *augmenting path (cycle)* is a simple path (cycle) in $G_f$. The *cost of a path (cycle)* is the sum of the costs of all edges on the path (cycle).

## 2.3 Optimality and Approximate Optimality

In this section we define the notion of an $\epsilon$-optimal pseudoflow and show that for $\epsilon < 1/n$, an $\epsilon$-optimal circulation is optimal.

The following theorem of Ford and Fulkerson [19] provides an optimality criterion for a circulation.

**Theorem 2.3.1** *A circulation $f$ is a minimum-cost circulation if and only if there exists a price function $p$ such that $\forall (v,w) \subset V \times V$,*

$$c_p(v,w) > 0 \Rightarrow f(v,w) = l(v,w) \tag{2.9}$$

*and*

$$c_p(v,w) < 0 \Rightarrow f(v,w) = u(v,w). \tag{2.10}$$

The optimality conditions (2.9) and (2.10) are called *complementary slackness* conditions, and an edge $(v,w)$ satisfying these conditions is said to be *in kilter*. We use the term kilter because of the relationship between our algorithm and the out-of-kilter method [57,24]. Figure 2.1a shows a *kilter diagram* [52], which is a pictorial representation of the complementary slackness conditions.

The antisymmetry constraints (2.2), (2.5), and (2.7) imply that an edge $(v,w)$ is in kilter if and only if the corresponding edge $(w,v)$ is in kilter.

The following theorem [19] gives an optimality criterion that does not involve the price function.

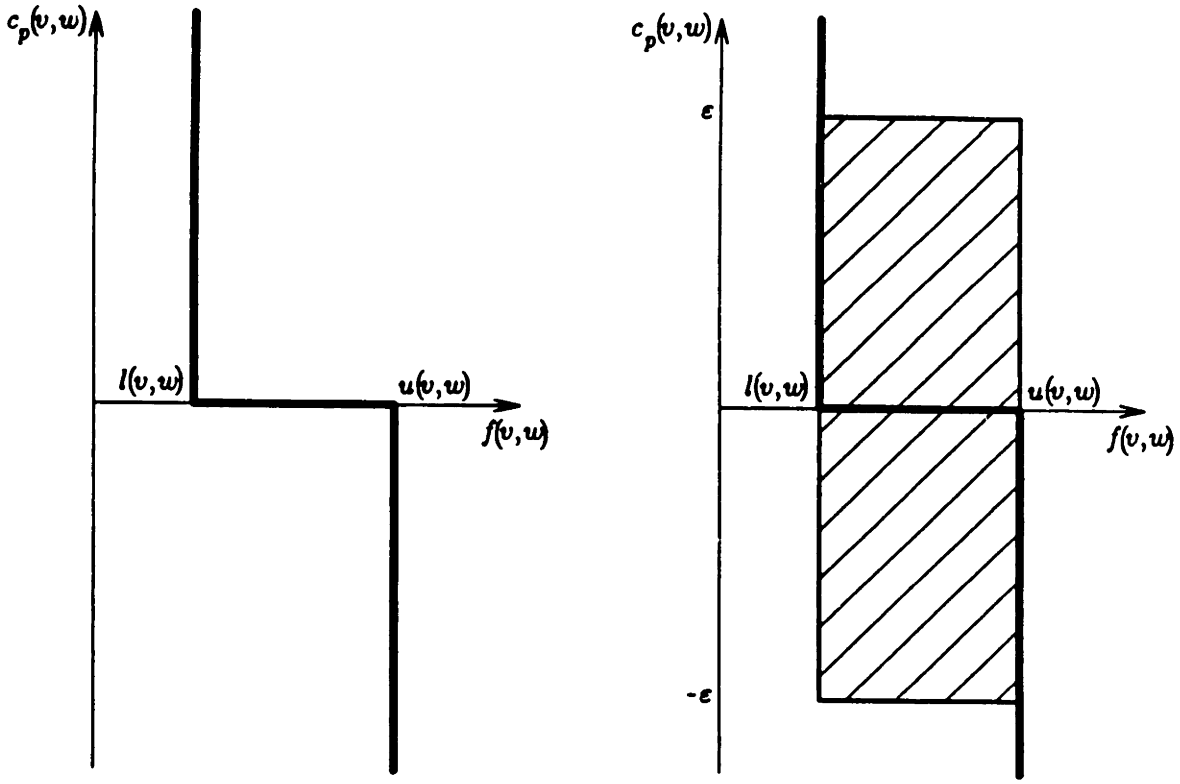
**Theorem 2.3.2** *A circulation $f$ is optimal if and only if the residual graph $G_f$ contains no cycles of negative cost.*

To define $\epsilon$-optimality, we use the notion of $\epsilon$-relaxation of the complementary slackness conditions [7,70]. Given $\epsilon \geq 0$, we say that a pseudoflow $f$ is $\epsilon$-optimal if the following relaxations of the complementary slackness conditions hold: $\forall (v,w) \in V \times V$,

$$c_p(v,w) > \epsilon \Rightarrow f(v,w) = l(v,w) \tag{2.11}$$

and

$$c_p(v,w) < -\epsilon \Rightarrow f(v,w) = u(v,w). \tag{2.12}$$




(a) Kilter diagram (b) Approximate optimality

Figure 2.1:
(a) Kilter diagram. If $f$ is an optimal circulation, the edge $(v, w)$ is on the heavy curve.
(b) $\epsilon$-optimality. If $f$ is $\epsilon$-optimal, the edge $(v, w)$ can be in the shaded rectangle as well as on the heavy curve.

We say that a pseudoflow $f$ is *$\epsilon$-optimal* if there exists a price function $p$ with respect to which the pseudoflow $f$ is $\epsilon$-optimal. Figure 2.1b illustrates the concept of $\epsilon$-optimality in terms of kilter diagrams.

The antisymmetry constraints (2.2), (2.5), and (2.7) imply the following lemma:

**Lemma 2.3.3** *An edge $(v, w)$ satisfies conditions (2.11) and (2.12) if and only if the corresponding edge $(w, v)$ satisfies conditions (2.11) and (2.12).*

The following simple fact is used extensively in our presentation.

**Lemma 2.3.4** *Suppose that pseudoflow $f$ is $\epsilon$-optimal with respect to a price function $p$. Then for any residual edge $(v, w)$ we have $c_p(v, w) \geq -\epsilon$ (i.e., the cost of a residual edge cannot be too small).*

The following theorem of Bertsekas [6] shows that when $\epsilon$ is small enough, an $\epsilon$-optimal circulation is optimal.

**Theorem 2.3.5** *Assume that all edge costs are integers. Then for any $0 \leq \epsilon < 1/n$, an $\epsilon$-optimal circulation $f$ is optimal.*

*Proof*: Consider a cycle in $G_f$. By Lemma 2.3.4, the $\epsilon$-optimality of $f$ implies that the cost of the cycle is at least $-n\epsilon$, which is greater than $-1$. Since the costs are integers, the cost of the cycle must be at least 0. The theorem then follows from the optimality criterion given by Theorem 2.3.2. ∎

## 2.4 High-Level Description of the Algorithm

Theorem 2.3.5 suggests the algorithm *Min-Cost* summarized in Figure 2.2. First, the algorithm finds a circulation (using a maximum flow algorithm) and sets the prices of all vertices to zero. The resulting circulation is $C$-optimal (recall that $C$ is the largest absolute value of an edge cost). Then, the algorithm iteratively improves the approximation (using the *Improve-Approximation* subroutine), until the error becomes less than $1/n$. At this point, the current solution is optimal.

*Remark*: The algorithm need not use a maximum flow subroutine in the initialization stage. It can start with any pseudoflow. If the problem is infeasible, we can discover this fact during the first execution of *Improve-Approximation* because the increase in vertex prices will be greater than allowed by the analysis below. We use the maximum flow subroutine only to be able to assume, without loss of generality, that the input problem is feasible, so that we do not have to worry about feasibility in our presentation.

**Procedure** *Min-Cost*$(V, E, l, u, c)$;

$\epsilon \leftarrow \max_{(v,w)\in E} |c(v,w)|$;
$f_\epsilon \leftarrow$ feasible circulation; ⟨⟨ use a maximum flow subroutine ⟩⟩
$\forall v$, $p_\epsilon(v) \leftarrow 0$;
**while** $\epsilon \geq 1/n$ **do**
    $\langle f_{\epsilon/2}, p_{\epsilon/2}\rangle \leftarrow$ *Improve-Approximation*$(f_\epsilon, p_\epsilon, \epsilon)$;
    $\epsilon \leftarrow \epsilon/2$;
**end**;
**return**$(f_\epsilon)$;

**end.**

Figure 2.2: The minimum-cost flow algorithm.

**Theorem 2.4.1** *Let $D(n,m)$ be the running time of the Improve-Approximation subroutine. Then the minimum-cost flow algorithm runs in $O(D(n,m)\log(nC))$ time and returns a minimum-cost flow.*

*Proof*: Immediate from Theorem 2.3.5 and the above discussion. ∎

The inputs to the *Improve-Approximation* subroutine are a circulation $f_\epsilon$, a price function $p_\epsilon$, and an error parameter $\epsilon$, such that $f_\epsilon$ is $\epsilon$-optimal with respect to $p_\epsilon$. The outputs of the subroutine are a circulation $f_{\epsilon/2}$ and a price function $p_{\epsilon/2}$ such that $f_{\epsilon/2}$ is $(\epsilon/2)$-optimal with respect to $p_{\epsilon/2}$. The subroutine sets the flow through every out-of-kilter edge to the upper or lower bound to bring the edge in kilter. The resulting pseudoflow is $(\epsilon/2)$-optimal (in fact, 0-optimal), but the conservation constraints (2.6) may be violated. Then, the pseudoflow is transformed into a circulation by applying a sequence of operations that preserves $(\epsilon/2)$-optimality. This transformation is done using generalizations of maximum flow techniques. As we shall see, both the techniques described in Chapter 1 and the techniques of Dinic [14] can be used.

The cost scaling algorithms of Röck and of Bland and Jensen use a maximum flow algorithm in the inner loop. Since the *Improve-Approximation* subroutine is a

generalization of a maximum flow algorithm, our minimum-cost flow algorithm is similar to the cost scaling algorithms. The cost-scaling algorithms halve the error in $O(n)$ iterations of the inner loop, however, whereas our algorithm halves the the error in approximation in a single iteration.

Intuitively, the algorithm can be viewed as simulated annealing [48]. At the beginning of each iteration, we release the potential energy of the system, and then let the system cool with the speed determined by $\epsilon$. When $\epsilon$ is big, the energy is dissipated quickly, but we are unlikely to find an optimal solution. When $\epsilon$ is small enough, we are guaranteed to find the optimal solution. At each iteration, the value of $\epsilon$ is selected so that the system cools reasonably quickly.

## 2.5 Generic Improve-Approximation Subroutine

The generic *Improve-Approximation* subroutine is a generalization of the generic maximum-flow algorithm of Section 1.2. An implementation of the generic subroutine using the *discharge* operation as described in Section 2.7 is almost identical to the earlier minimum-cost flow algorithm of Bertsekas [6], but we use the subroutine in a different way. In our algorithm the subroutine is used to reduce the approximation parameter $\epsilon$ by a factor of two at each iteration. In the algorithm of Bertsekas, the value of $\epsilon$ is set to $1/(n+1)$ at the initialization, so the algorithm terminates in one iteration — but may take exponential time.

Our presentation and analysis of the generic subroutine and its implementations follow the presentation and analysis of the generic maximum flow algorithm. Although the generalization of the algorithms is straight-forward, the generalization of the analysis is more complicated. In fact, some time bounds that we prove in this chapter are not quite as good as the corresponding bounds for the maximum flow algorithms. The differences are discussed in Section 2.7.

The generic *Improve-Approximation* subroutine is described in Figure 2.3. The subroutine forces all edges in kilter and then transforms the resulting pseudoflow into an $(\epsilon/2)$-optimal circulation. The *basic operations* used to manipulate the

**Procedure** *Improve-Approximation*$(f, p, \epsilon)$;

⟨⟨ initialization⟩⟩
$\forall (v, w) \in V \times V$ **do begin**
    **if** $c_p(v, w) > 0$ **then** $f(v, w) \leftarrow l(v, w)$;
    **if** $c_p(v, w) < 0$ **then** $f(v, w) \leftarrow u(v, w)$;
**end;**
⟨⟨ loop⟩⟩
**while** ∃ a basic operation that applies
    select a basic operation and apply it;
**return**$(f, p)$;

**end.**

Figure 2.3: The generic Improve-Approximation subroutine. The running time of the subroutine depends on the order in which basic operations are selected and on details of implementation.

pseudoflow and the prices are *push* and *update-price*. The operations are described in Figure 2.4 and are illustrated in terms of kilter diagrams in Figure 2.5. A *push* through an edge $(v, w)$ increases $f(v, w)$ and $b_f(w)$ by at most $\delta = \min(b_f(v), r_f(v, w))$ and decreases $f(w, v)$ and $b_f(v)$ by the same amount. For the purpose of the analysis, we distinguish between saturating and nonsaturating pushes. A push is *saturating* if $r_f(v, w) = 0$ after the push and *nonsaturating* otherwise. The *update-price* operation sets the price of a vertex $v$ to the highest value allowed by the $(\epsilon/2)$-optimality constraints.

The following lemmas give important properties of the *push* and *update-price* operations.

**Lemma 2.5.1** *A pushing operation preserves the* $(\epsilon/2)$*-optimality of a pseudoflow.*

*Proof*: Let $f$ be an $(\epsilon/2)$-optimal pseudoflow with respect to a price function $p$. Suppose a pushing operation is applied to an edge $(v, w)$. Since $(v, w) \in E_f$ and a pushing operation is applicable to $(v, w)$, it follows that $-\epsilon/2 \leq c_p(v, w) < -\epsilon/4$. Note

*Push*$(v, w)$.
**Applicability:** $v$ is active, $r_f(v, w) > 0$, **and** $c_p(v, w) < -\epsilon/4$.
**Action:** Send $\delta = \min(b_f(v), r_f(v, w))$ units of flow from $v$ to $w$.

*Update-Price(v).*
**Applicability:** $v$ is active **and** $\forall w \in V\ r_f(v, w) > 0 \Rightarrow c_p(v, w) \geq -\epsilon/4$.
**Action:** Replace $p(v)$ by $\min_{(v,w)\in E_f}(p(w) + c(v, w) + \epsilon/2)$.

Figure 2.4: Push and update-price operations.

that the relaxed complementary slackness conditions do not restrict flow through edges of small cost, therefore changing the flow through any residual edge $(v, w)$ with $|c_p(v, w)| \leq \epsilon/2$ preserves $(\epsilon/2)$-optimality. It follows that a pushing operation preserves $(\epsilon/2)$-optimality. ∎

**Lemma 2.5.2** *Suppose $f$ is an $(\epsilon/2)$-optimal pseudoflow with respect to a price function $p$ and a price updating operation is applied to a vertex $v$. Then the price of $v$ increases by at least $\epsilon/4$ and the pseudoflow $f$ is $(\epsilon/2)$-optimal with respect to the resulting price function $p'$.*

*Proof*: First we prove that the price of $v$ increases by at least $\epsilon/4$. The price updating operation changes the price of the vertex $v$ from $p(v)$ to $p'(v)$ and does not affect prices of other vertices. Since a price updating operation is applicable to $v$, we have $c(v, w) - p(v) + p(w) \geq -\epsilon/4$ for all residual edges $(v, w)$. Therefore, we have $c(v, w) + p(w) + \epsilon/2 \geq p(v) + \epsilon/4$ for all edges $(v, w) \in E_f$, so the new price $p'(v) \geq p(v) + \epsilon/4$ by definition of the price updating operation.

Now we prove the second claim of the lemma. Because of the antisymmetry, it is enough to show that for all $w \in V$, the relaxed complementary slackness conditions corresponding to $(v, w)$ remain valid. The constraint 2.11 holds after the price update because it holds before the price update and from the first part of the proof we know that $c_{p'}(v, w) < c_p(v, w)$. The constraint 2.12 clearly holds

if $f(v,w) = u(v,w)$. If $f(v,w) < u(v,w)$ then we have $(v,w) \in E_f$ so $c(v,w) - p'(v) + p'(w) = c(v,w) - p'(v) + p(w) \geq -\epsilon/2$ by the definition of the price updating operation. Therefore the constraint 2.12 also holds in this case. ∎

**Lemma 2.5.3** *If a pseudoflow $f$ is $(\epsilon/2)$-optimal with respect to a price function $p$ and $v$ is an active vertex, then either a pushing or a price updating operation is applicable to $v$.*

*Proof*: Suppose $v$ is active and no pushing operation is applicable to $v$. By $(\epsilon/2)$-optimality of $f$, for any residual edge $(v,w)$, we have $c_p(v,w) \geq -\epsilon/2$ by Lemma 2.3.4. Furthermore, since the edge $(v,w)$ cannot be used for pushing, we have $c_p(v,w) \geq -\epsilon/4$. Therefore a price updating operation is applicable. ∎

The generic version of *Improve-Approximation*, described in Figure 2.3, repetitively performs an applicable basic operation on the current pseudoflow $f$ and price function $p$ until no basic operation applies. We shall prove that the generic version of *Improve-Approximation* is correct if it terminates, and then we shall prove termination.

**Theorem 2.5.4** *If the generic Improve-Approximation subroutine terminates on an $\epsilon$-optimal input circulation, then the pseudoflow $f$ output by the subroutine is an $(\epsilon/2)$-optimal circulation.*

*Proof*: The subroutine terminates when there are no vertices with positive balance and therefore no vertices with negative balance, so $f$ is a circulation. The $(\epsilon/2)$-optimality of $f$ follows from Lemmas 2.5.1 and 2.5.2 and the fact that the pseudoflow computed during the initialization step of *Improve-Approximation* is $(\epsilon/2)$-optimal (in fact, 0-optimal). ∎

We conclude this section with a remark that shows how to find an optimal circulation given an optimal price function (a price function $p$ is *optimal* if there is a circulation $f$ that satisfies the complementary slackness conditions 2.9 and 2.10). Suppose that we force the edges in kilter in the same way as in the initialization

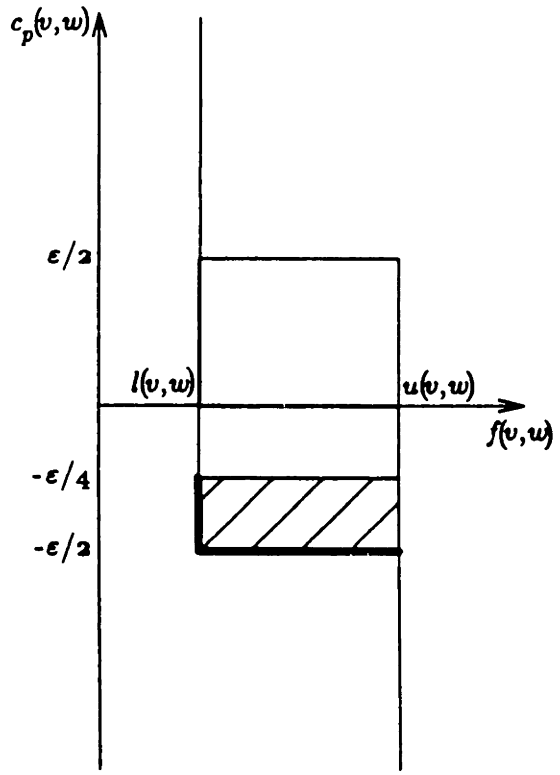



Figure 2.5: Push$(v, w)$ is allowed if $b_f(v) > 0$ and $(v, w)$ is in the shaded rectangle (including the heavy curve). If $b_f(v) > 0$ but no push operation is applicable, Update-Price$(v)$ is applicable. Increasing the price of $v$ corresponds to shifting the kilter diagram down. The absence of edges in the shaded rectangle guarantees that the price of $v$ increases by at least $\epsilon/4$.

step of the generic *Improve-Approximation* subroutine. It can be shown using the techniques of the next section that the resulting pseudoflow can be converted into a circulation by changing the flow only through edges with zero reduced cost. The resulting circulation satisfies the complementary slackness conditions and therefore is optimal. This circulation can be found in one maximum flow computation.

## 2.6 Analysis of the Generic Subroutine

In this section we analyze the generic *Improve-Approximation* subroutine. The analysis is similar to the analysis of the generic maximum flow algorithm. We start the running time analysis by bounding the amount of increase in vertex prices

during an execution of the generic subroutine.

**Lemma 2.6.1** *Let $f$ be a pseudoflow and let $f'$ be a circulation. Then for any vertex $v$ with a positive balance $b_f(v)$, there exists a vertex $w$ with a negative balance $b_f(w)$ and a sequence of vertices $u_1, \ldots, u_{l-1}$ such that $(v, u_1, \ldots, u_{l-1}, w)$ is a simple path in $G_f$ and $(w, u_{l-1}, \ldots, u_1, v)$ is a simple path in $G_{f'}$.*

*Proof*: Fix a vertex $v$ with a positive balance. Let $G_+ = (V, E_+)$ where $E_+ = \{(i,j) | f'(i,j) > f(i,j)\}$ and let $G_- = (V, E_-)$ where $E_- = \{(i,j) | f'(i,j) < f(i,j)\}$. This definition implies that $E_+ \subseteq E_f$, because for any edge $(i,j) \in E_f$ we have $f(i,j) < f'(i,j) \leq u(i,j)$. Similarly, we have $E_- \subseteq E_{f'}$. Furthermore, if $(i,j)$ is an edge in $G_+$, then $(j,i)$ is an edge in $G_-$ by antisymmetry. Therefore it is enough to show that if $b_f(v) > 0$, then there is a simple path $(v, u_1, \ldots, u_{l-1}, w)$ in $G_+$ such that $b_f(w) < 0$.

Assume by the way of contradiction that such a path does not exist, i.e. all vertices reachable from $v$ in $G_+$ have nonnegative balance. Let $S$ be the set of all vertices reachable from $v$ in $G_+$ and let $\overline{S} = V - S$. The vertices in $S$ have nonnegative balance, and since $v \in S$ and $b_f(v) > 0$, we have $\sum_{j \in S} b(j) > 0$. Also, for every pair of vertices $(i,j) \in S \times \overline{S}$, we have $f(i,j) \geq f'(i,j)$, for otherwise we would have $j \in S$.

Now we obtain a contradiction as follows.

$$\begin{aligned}
0 &= \textstyle\sum_{(i,j) \in S \times \overline{S}} f'(i,j) && (f' \text{ is a circulation}) \\
&\leq \textstyle\sum_{(i,j) \in S \times \overline{S}} f(i,j) \\
&= \textstyle\sum_{(i,j) \in S \times \overline{S}} f(i,j) + \sum_{(i,j) \in S \times S} f(i,j) && (\text{antisymmetry}) \\
&= \textstyle\sum_{(i,j) \in S \times V} f(i,j) \\
&= -\textstyle\sum_{j \in S} b_f(j) && (\text{definition of balance}) \\
&< 0
\end{aligned}$$

∎

The following key lemma bounds the amount of increase in prices during an execution of the subroutine. This lemma is similar to the lemma of Bland and

Jensen [8] that bounds the number of maximum flow computations in a scaling step.

**Lemma 2.6.2** *The price of any vertex $v$ increases by at most $(3/2)n\epsilon$ during an execution of Improve-Approximation.*

*Proof*: Since we change vertex prices only by using price updating operations, which apply only to vertices with positive balance, it is enough to prove the lemma for a vertex $v$ that has a positive balance at some time during the execution of *Improve-Approximation*. Let $f$ and $p$ be the $(\epsilon/2)$-optimal pseudoflow and price function at this point. Recall that $f_\epsilon$ and $p_\epsilon$ are the $\epsilon$-optimal circulation and the price function at the beginning of the execution of *Improve-Approximation*. Let $v, u_1, \ldots, u_{l-1}, w$ be a sequence of vertices satisfying Lemma 2.6.1. Applying the $\epsilon$-optimality conditions to the edges on the path $P_{f_\epsilon} = (w, u_{l-1}, \ldots, u_1, v)$ in $G_{f_\epsilon}$, we obtain

$$p_\epsilon(w) \leq p_\epsilon(v) + l\epsilon + \sum_{(i,j) \text{ on } P_{f_\epsilon}} c(i,j). \tag{2.13}$$

Applying the $(\epsilon/2)$-optimality conditions to the edges on the path $P_f = (v, u_1, \ldots, u_{l-1}, w)$ in $G_f$, we obtain

$$\begin{array}{rcl} p(v) & \leq & p(w) + l(\epsilon/2) + \sum_{(j,i) \text{ on } P_f} c(j,i) \\ & = & p(w) + l(\epsilon/2) - \sum_{(i,j) \text{ on } P_{f_\epsilon}} c(i,j). \end{array} \tag{2.14}$$

By Lemma 2.6.1, the balance $b_f(w)$ is negative. Note that application of basic operations cannot create a new vertex with a negative balance. Thus $w$ has had a negative balance after the initialization step of *Improve-Approximation*. The prices of vertices with negative balance do not change, so $p(w) = p_\epsilon(w)$. Combining this observation with inequalities 2.13 and 2.14, we get $p(v) \leq p_\epsilon(v) + (3/2)l\epsilon \leq p_\epsilon(v) + (3/2)n\epsilon$. ∎

*Remark*: Although Lemmas 2.6.1 and 2.6.2 are sufficient for our presentation, the following statement gives additional insight into the problem. This statement can

be proven in a way similar to Lemmas 2.6.1 and 2.6.2. *Let $f_1$ be an $\epsilon_1$-optimal circulation with respect to a price function $p_1$ and let $f_2$ be an $\epsilon_2$-optimal circulation with respect to a price function $p_2$. Let $S$ be a set of vertices such that for all $s \in S$ we have $p_1(s) = p_2(s)$ and any vertex $v \in V$ is reachable from some vertex $s \in S$ in $G_{f_1}$ or in $G_{f_2}$. Then for any vertex $v$, we have $|p_1(v) - p_2(v)| \leq (\epsilon_1 + \epsilon_2)n$.*

**Lemma 2.6.3** *The number of the price updates during an execution of Improve-Approximation is at most $6n^2$.*

*Proof*: Immediate from lemmas 2.5.2 and 2.6.2. ∎

Lemmas 2.6.2 and 2.6.3 enable us to amortize the operations performed by the algorithm over increases in vertex prices.

**Lemma 2.6.4** *The number of the saturating push operations during an execution of Improve-Approximation is at most $7nm$.*

*Proof*: For any undirected edge $\{v, w\}$, consider saturating pushes from $v$ to $w$ and from $w$ to $v$. Consider a saturating push from $v$ to $w$. In order to push flow from $v$ to $w$ again, the subroutine must first push from $w$ to $v$, which can not happen until the price of $w$ increases by at least $\epsilon/2$. Similarly, $p(v)$ must increase by at least $\epsilon/2$ between saturating pushes from $w$ to $v$. By charging all saturating pushes from $v$ to $w$ (except for the first one) to price increases of $v$ and applying Lemma 2.6.2, we can bound the number of pushes that use the undirected edge $\{v, w\}$ by $2 + 6n \leq 7n$ (assuming $n \geq 2$). Summing over all undirected edges gives the desired bound. ∎

At this point we introduce the concepts of an *admissible edge* and the *admissible graph*. Given a pseudoflow $f$ and a price function $p$, we say that an edge $(v, w)$ is admissible if $(v, w) \in E_f$ and $c_p(v, w) < -\epsilon/4$. Note that if $(v, w)$ is an admissible edge and $v$ is active, then a pushing operation is applicable to $(v, w)$. For a given price function $p$, the admissible graph $G_A$ is the graph of all admissible edges: $G_A = (V, E_A)$, where $E_A = \{(v, w) \in E_f | c_p(v, w) < -\epsilon/4\}$.

The following lemma is due to Robert Tarjan.

**Lemma 2.6.5** *At any time during the execution of the generic Improve-Approximation subroutine, the admissible graph $G_A$ is acyclic.*

*Proof*: We prove the lemma by induction on the number of basic operations. For the basis, note that after the initialization there are no admissible edges, because all edges in the residual graph have nonnegative costs. Now suppose that $G_A$ is acyclic before a basic operation is applied. A pushing operation can delete an admissible edge but cannot create a new admissible edge. Therefore a pushing operation cannot create a cycle in $G_A$.

Now consider a price updating operation applied to a vertex $v$. This operation affects only admissible edges adjacent to $v$, so it is enough to show that after the operation there is no cycle in $G_A$ that passes through $v$. Before the price update, the reduced cost of any residual edge entering $v$ is at least $-\epsilon/2$. The price update increases this cost by at least $\epsilon/4$. Therefore after the price update, there are no admissible edges going into $v$. It follows that after the price updating operation, there is no cycle in $G_A$ that passes through $v$. ∎

The following lemma bounds the number of nonsaturating pushes. This lemma has been strengthened to its current form with the help of Ron Rivest.

**Lemma 2.6.6** *The number of the nonsaturating pushing operations in an execution of Improve-Approximation is $O(n^2m)$.*

*Proof*: For the purpose of the proof, we define a function $h$ that maps vertices into real numbers. For a vertex $v$, let $h(v)$ be 0 if the out-degree of $v$ in the admissible graph is zero. Otherwise, let $h(v)$ be the minimum, over all paths $P$ in the admissible graph that start at $v$, of the sum of reduced costs of edges in $P$. Define $\Phi = 0$ if there are no active vertices and $\Phi = \sum_{\{v|v \text{ is active}\}} h(v)$ otherwise. Note that for any vertex $v$, we have $-(\epsilon/2)n \leq h(v) \leq 0$ and therefore $-(\epsilon/2)n^2 \leq \Phi \leq 0$. Since the admissible graph is acyclic, each nonsaturating pushing operation causes $\Phi$ to increase by at least $\epsilon/4$.

Next we bound the total amount of decrease in $\Phi$. A saturating pushing operation causes $\Phi$ to decrease by at most $(\epsilon/2)n$. The total decrease in $\Phi$ due to saturating pushes is at most $7n^2m(\epsilon/2)$ by Lemma 2.6.4. As we have seen in the proof of Lemma 2.6.5, a price updating operation applied to a vertex $v$ removes all admissible edges going into $v$. Therefore, such an operation can decrease $h(v)$, but cannot increase $h(w)$ for any $w \neq v$. It follows that each price update causes $\Phi$ to decrease by at most $(\epsilon/2)n$. The total amount of decrease in $\Phi$ due to price updates is at most $6n^3(\epsilon/2)$ by Lemma 2.6.3. After the initialization step, $\Phi$ is at most $n^2(\epsilon/2)$, and $\Phi$ is always nonnegative. Therefore the number of nonsaturating pushes is at most $14n^2m + 12n^3 + 2n^2 = O(n^2m)$. ∎

The following theorem bounds the number of basic operations in the generic *Improve-Approximation* subroutine.

**Theorem 2.6.7** *The generic Improve-Approximation subroutine terminates after $O(n^2m)$ basic operations.*

*Proof*: Immediate from Lemmas 2.5.1, 2.5.2, 2.6.3, 2.6.4, and 2.6.6. ∎

As we know, the maximum flow problem is a special case of the minimum-cost circulation problem. The standard reduction transforms an instance of the maximum flow problem into an instance of the minimum-cost flow problem as follows. Lower bounds and costs of all edges are set to zero. Then, a new edge $(t, s)$ going from the sink $t$ to the source $s$ is introduced. The upper bound on the flow through this edge is set to the sum of capacities of all edges going out of the source, the lower bound on the flow is set to zero, and the cost of the edge is set to $-n$. Since this new edge is the only edge with negative cost and all other edges have zero cost, an optimal circulation maximizes flow through this edge — and maximizes flow from the source to the sink in the original network. If we saturate the edge $(t, s)$, the resulting pseudoflow is 1-optimal with respect to the zero price function. The generic *Improve-Approximation* subroutine applied at this point works almost in the same way as the generic maximum flow algorithm. The only difference is that the subroutine changes the price of the source, whereas the maximum flow algorithm

does not change the distance label of the source. This difference is minor, because the maximum flow algorithm can be easily modified to allow changes in the distance label of the source [37].

We would like to conclude this section with a discussion of the choice of the value $-\epsilon/4$ as an upper bound on the reduced cost of admissible edges. There are other possible choices for this lower bound. In particular, we can set this bound to zero, allowing pushes through residual edges with a negative reduced cost and modifying the price updating operation to apply to a vertex $v$ only when reduced costs of all residual edges going out of $v$ are nonnegative. The asymptotic complexity bounds stated in this chapter on the performance of the generic subroutine and its derivatives still apply if we make this change in the algorithm (although some proofs and constant factors change).

The justification for the choice of the value $-\epsilon/4$ is on purely intuitive level. We can define the *cost of a pseudoflow* by equation (2.8). The choice of the value $-\epsilon/4$ (or $-k\epsilon$ for a constant $k$ such that $0 < k < 1/2$) assures that moving a unit of flow by a pushing operation changes the cost of a pseudoflow by at least $-\epsilon/4$ (or $-k\epsilon$ times the number of flow units pushed). This change in the cost of pseudoflow seems important for the validity of Conjecture 1 stated in the next section and for potential generalization of the Ahuja-Orlin algorithm [1] to the minimum-cost flow problem.

## 2.7 Sequential Implementation

The running time of the generic subroutine depends upon the order in which the basic operations are applied and on the details of implementation, but it is clear that any reasonable sequential implementation will run in polynomial time. As a first step toward obtaining an efficient sequential implementation of the generic subroutine, we shall describe a simple refinement of the subroutine that runs in $O(n^2m)$ time. Then, we describe a first-in, first-out version of the generic subroutine, which is similar to the first-in, first-out maximum flow algorithm described in

*Push/update*($v$).
**Applicability:** $v$ is active.
**Action:** Let $\{v, w\}$ be the current edge of $v$.
**If** Push($v, w$) is applicable, apply it;
**else**
**if** $\{v, w\}$ is not the last edge on the edge list of $v$ **then**
replace $\{v, w\}$ as the current edge of $v$
by the next edge on the edge list of $v$;
**else begin**
make the first edge on the edge list of $v$ the current edge;
apply *Update-Price*($v$);
**end.**

Figure 2.6: The push/update operation.

section 1.4. Unlike in the maximum flow case, however, we are not able to prove the $O(n^3)$ time bound on the first-in, first-out algorithm (although we believe that this bound holds). We shall show a different implementation of the generic *Improve-Approximation* subroutine that runs in $O(n^3)$ time. In the next section, we shall describe implementation of the generic subroutine that uses the dynamic tree data structure and runs in $O(nm \log n)$ time.

We need some data structures to represent the network and the pseudoflow. We associate a positive direction and the following four values with each undirected edge $\{v, w\}$: $l(v, w)$, $u(v, w)$, $c(v, w)$, and $f(v, w)$. Each vertex $v$ has a list of the incident undirected edges $\{v, w\}$ in a fixed order. Each edge $\{v, w\}$ appears in exactly two lists, the one for $v$ and the one for $w$. Each vertex $v$ has a balance $b_f(v)$ and a *current edge* $\{v, w\}$, which is the current candidate for a push out of $v$. Initially the current edge is the first edge on the edge list of $v$. After the initialization, the refined subroutine applies the *push/update* operation described in Figure 2.6 until there are no active vertices.

We need to show that *push/update* uses the price update operation correctly.

**Lemma 2.7.1** *The push/update procedure uses a price updating operation only*

*when this operation is applicable.*

*Proof*: The *push/update* procedure applies the price updating operation only to active vertices with positive balance. Just before the price update, for each edge $(v, w)$ either $c_p(v, w) \geq -\epsilon/4$ or $r_f(v, w) = 0$, because $p(v)$ has not changed since $\{v, w\}$ was a current edge, $r_f(v, w)$ cannot increase unless $c_p(v, w) > -\epsilon/4$, and $p(w)$ never decreases. The lemma follows from the definition of the price updating operation. ∎

The refined subroutine needs one additional data structure, a set $Q$ containing all vertices with positive balance. Initially $Q$ contains vertices whose balance has been made positive during the initialization. Maintaining $Q$ takes only $O(1)$ time per *push/update* operation. (Such an operation applied to an edge $\{v, w\}$ may require adding $w$ to $Q$ and/or deleting $v$.)

**Theorem 2.7.2** *The refined algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating pushing step, for a total of $O(n^2m)$ time.*

*Proof*: Let $v \in V$ and let $\Delta_v$ be the number of edges on the edge list of $v$. A price update of $v$ requires a single scan of the edge list of $v$. By the proof like that of Lemma 2.6.3, the total number of passes through the edge list of $v$ is at most $12n + 1$: one for each of at most $6n$ price updates of $v$, one before each price update as the current edge runs through the list, and one after the last price update. Every *push/update* operation selecting $v$ either causes a push, changes the current edge of $v$, or increases $p(v)$. The total time spent in *push/update* operations selecting $v$ is $O(n\Delta_v)$ plus $O(1)$ time per push out of $v$. Summing over all vertices and applying Lemmas 2.6.4 and 2.6.6 gives the theorem. ∎

As in the maximum flow case, our next step is to describe the *first-in, first-out* version of the generic subroutine. This version of the subroutine maintains $Q$ as a queue and applies *discharge* operations until the queue is empty. The *discharge* operation, described in Figure 2.7, is the same as the maximum flow algorithm except that the *push/update* operation is used instead of the *push/relabel* operation.

*Discharge.*
Applicability: $Q \neq \emptyset$.
Action: Remove the vertex $v$ on the front of $Q$.
(Vertex $v$ must be active.)
**Repeat**
apply *push/update*($v$);
**if** $w$ becomes active during this *push/update* operation **then**
add $w$ to the rear of $Q$;
**until** $b_f(v) = 0$ **or** $p(v)$ increases.
**If** $v$ is still active **then** add $v$ to the rear of $Q$.

Figure 2.7: The discharge operation.

We define *passes* over the queue in the same way as in Section 1.4. Pass one consists of the discharging operations applied to the vertices added to the queue during the initialization. Given that pass $i$ is defined, pass $i + 1$ consists of the discharging operations applied to vertices on the queue that were added during pass $i$.

**Lemma 2.7.3** *The number of passes over the queue in the execution of the first-in, first-out Improve-Approximation subroutine is* $O(n^3)$.

*Proof*: Define the function $h$ in the same way as in the proof of Lemma 2.6.6, and define $\Phi = 0$ if $f$ is a circulation and $\Phi = \min_{\{v|b(v)>0\}} h(v)$ otherwise. Recall that for any $v$, we have $-(\epsilon/2)n \leq h(v) \leq 0$ and therefore $-(\epsilon/2)n \leq \Phi \leq 0$. Consider the effect on $\Phi$ of a single pass over the queue. If the price function has not changed during the pass, $\Phi$ must increase by at least $\epsilon/4$. If the price function changes during the pass, $\Phi$ decreases by at most $(\epsilon/2)n$. The total number of passes during which $\Phi$ decreases is at most $6n^2$ by Lemma 2.6.3, and the total amount of decrease in $\Phi$ is at most $3n^3\epsilon$. After the initialization step $\Phi$ is at least $-(\epsilon/2)n$, and at the end of the subroutine $\Phi$ is zero. The number of passes during which $\Phi$ increases is at most $12n^3 + 2n$, and the overall number of passes is $6n^2 + 12n^3 + 2n = O(n^3)$.
∎

Note that the bound given by the above lemma is not as good as the corresponding bound of $O(n^2)$ on the number of passes for the maximum flow algorithm. We suspect that the analysis given in Lemma 2.7.3 can be improved to match the corresponding maximum flow bound to within a constant factor.

**Conjecture 1** *The bounds given by Lemma 2.7.3 can be improved to $O(n^2)$, either for the described version of the first-in, first out algorithm or for a modified version.*

All implementations of the maximum flow algorithm generalize to the minimum cost flow case. The bounds that we are able to prove for the implementations of the generic *Improve-Approximation* subroutine that are based on the first-in, first-out strategy, however, are worse then the corresponding maximum flow bounds. Conjecture 1 would imply the same asymptotic bounds for all variations of the *Improve-Approximation* subroutine based on the first-in, first-out strategy as for the corresponding variations of the maximum flow algorithm. In particular, the conjecture would imply an $O(nm \log(n^2/m))$ sequential running time bound and an $O(n^2)$ bound on the number of parallel pulses.

The $O(n^3)$ bound on the number of passes of the first-in, first-out algorithm is not strong enough to prove an $O(n^3)$ time bound on the *Improve-Approximation* subroutine. Next we show a different implementation of the subroutine that runs in $O(n^3)$ time. This implementation is due to Charles Leiserson. We call this implementation the *wave subroutine* because of its similarity to the maximum flow algorithm described in [72]. When applied to the generic maximum flow algorithm of Section 1.2, the wave method yields an $O(n^3)$ time algorithm as well.

Unlike the previous implementations we have discussed, the wave implementation does not maintain a queue of active vertices. The implementation maintains a list $L$ of all vertices instead, and preserves the invariant that the vertices on the list are topologically ordered with respect to the admissible graph $G_A$ (i.e., for any two distinct vertices $v$ and $w$, if $w$ is reachable from $v$ in $G_A$, then $w$ appears after $v$ on the list).

*Discharge-2*($v$).
**Applicability:** $v$ is active.
**Action:** **Repeat**
Apply *push/update*($v$);
**until** $b_f(v) = 0$ **or** $p(v)$ increases.
**If** $p(v)$ has increased **then** move $v$ to the beginning of $L$;

Figure 2.8: The discharge-2 operation.

Initially, the list $L$ contains the vertices of $G$ in arbitrary order. The implementation makes passes over the list, applying the *discharge-2* operation, described in Figure 2.8, to active vertices. The *discharge-2* operation consists of applying the *push/relabel* operations to an active vertex until the excess becomes zero or the price of the vertex increases. In the latter case, the vertex is moved to the beginning of the list $L$, but the processing of the list $L$ continues from the previous position of this vertex. The subroutine terminates when there are no active vertices.

The key to the analysis of the wave subroutine is the observation that, because of the topological ordering of vertices on the list $L$, if no price update occurs during a pass over the vertex list, the subroutine terminates.

**Lemma 2.7.4** *The number of passes over the vertex list in the execution of the wave implementation of the Improve-Approximation subroutine is* $O(n^2)$.

*Proof*: First we show, by induction on the number of basic operations, that the wave subroutine maintains a topological ordering of the vertices with respect to the admissible graph $G_A$ (except in the middle of the *discharge-2* operation). The basis is trivial, because immediately after the initialization stage of the generic subroutine there are no admissible edges. A pushing operation preserves the topological ordering, because this operation cannot create a new admissible edge. Immediately after the price of a vertex is changed, the vertex is moved to the beginning of the list $L$. The resulting ordering is topological because a price updating operation applied to a vertex $v$ deletes all edges of $G_A$ going into $v$.

Next we show that if the vertex prices do not change during a pass over the vertex list, the subroutine terminates. This implies that there is at most one pass during which the price function does not change. The total number of passes is at most $6n^2+1$, because by Lemma 2.6.3, the number of passes during which the price of some vertex changes is at most $6n^2$.

Suppose that the price function does not change during a pass. Then no price updating operations are performed during this pass, and therefore the ordering of the vertices on the list $L$ does not change and every vertex is able to get rid of all of its excess. Since the vertices are processed in topological order, no vertex has excess after the pass, and the algorithm terminates. ∎

The $O(n^2)$ bound on the number of passes allows us to prove the $O(n^3)$ bound on the running time of the wave subroutine.

**Theorem 2.7.5** *The wave subroutine runs in $O(n^3)$ time.*

*Proof*: The work done by ***update-price*** and saturating ***push*** operations can be bounded by $O(nm)$ by a proof like that of Theorem 2.7.2. The number of non-saturating ***push*** operations is $O(n^3)$ because there are $O(n^2)$ passes and at most one nonsaturating push per vertex per pass. Finally, operations on the vertex list $L$ require $O(n)$ work per pass, for a total of $O(n^3)$ work. ∎

## 2.8 Use of Dynamic Trees

We improve the $O(n^3)$ bound on the *Improve-Approximation* subroutine given by Theorem 2.7.5 to $O(nm \log n)$ by using the dynamic tree data structure discussed in Section 1.5. The resulting subroutine is similar to the dynamic tree version of the maximum flow algorithm. Unlike the maximum flow algorithm, however, the subroutine implements a generic version that does not use the first-in, first-out ordering of operations, so we are not able to use the balancing of tree sizes.

*Send*($v$).
Applicability: $v$ is active.
Action: **While** *find-root*($v$) $\neq v$ **and** $b_f(v) > 0$ **do begin**
send $\delta = \min(b_f(v), \textit{find-value}(\textit{find-min}(v)))$ units of flow
along the tree path from $v$ by performing *change-value*($v, -\delta$);
**while** *find-value*(*find-min*($v$)) $= 0$ **do begin**
$u \leftarrow$ *find-min*($v$);
perform *cut*($u$) followed by *change-value*($u, \infty$);
**end;**
**end.**

Figure 2.9: The send operation.

As in the maximum flow implementation that uses the data structure, the edges of the dynamic trees are a subset of the current edges of the vertices. The current edge $\{v, w\}$ of a vertex $v$ is eligible to be a dynamic tree edge if the edge $(v, w)$ is admissible, but not all eligible edges are tree edges. The value $g(v)$ of a vertex $v$ in its dynamic tree is $r_f(v, \textit{parent}(v))$ if $v$ has a parent, $\infty$ if $v$ is a tree root. Initially each vertex is in a one-vertex dynamic tree and has value $\infty$.

By using appropriate tree operations we can push flow along an entire path in a tree, either causing a saturating push or moving flow excess from some vertex in the tree all the way to the tree root. The dynamic tree operations are charged to the price updates and saturating pushes in such a way that each price update and each saturating push is charged a constant number of times. Therefore the total number of charges is $O(nm)$ and, since each dynamic tree operation costs $O(\log n)$, the running time of the dynamic tree subroutine is $O(nm \log n)$.

The details of the improved subroutine, which we call the *dynamic tree subroutine*, are as follows. The heart of the subroutine is the procedure *send*($v$) defined in Figure 2.9, which pushes excess from a nonroot vertex $v$ to the root of its tree, cuts edges saturated by the push, and repeats these steps until $b_f(v) = 0$ or $v$ is a tree root.

At the top level, the dynamic tree subroutine is exactly the same as the simple

*Tree-Push/update*$(v)$.
*Applicability:* $v$ is an active tree root.
*Action:* Let $\{v, w\}$ be the current edge of $v$.
**(1) If** edge $(v, w)$ is admissible **then begin**
make $w$ the parent of $v$ by performing
*change-value*$(v, -\infty)$, *change-value*$(v, r_f(v, w))$, **and** *link*$(v, w)$;
push excess from $v$ by performing *send*$(v)$;
**end;**
**(2) else** $\langle\langle$ **edge** $(v, w)$ **is not admissible**$\rangle\rangle$
**(2a) if** $\{v, w\}$ is not the last edge on the edge list of $v$ **then**
replace $\{v, w\}$ as the current edge by the next edge on the list;
**(2b) else** $\langle\langle\{v, w\}$ **is the last edge on the edge list of** $v\ \rangle\rangle$ **begin**
make the first edge on the list the current one;
perform *cut*$(i)$ and *change-value*$(i)$ for every child $i$ of $v$;
apply a price updating operation to $v$;
**end.**

Figure 2.10: The tree-push/update operation.

implementation of the generic subroutine described in the previous section. However, we replace the *push/update* operation with the *tree-push/update operation* described in Figure 2.10.

A *tree-push/update* operation applies to a vertex $v$ with a positive balance that is the root of a dynamic tree. There are two main cases. The first case occurs if the current edge $\{v, w\}$ of $v$ is eligible for a pushing operation. In this case we link these trees by making $w$ the parent of $v$ and do a send operation from $v$. The second case occurs if the edge $\{v, w\}$ is not eligible for a pushing operation. In this case we update the current edge of $v$ and update the price of $v$ if necessary. If the price of $v$ is increased, we cut all tree edges entering $v$, to maintain the invariant that all dynamic tree edges are admissible.

It is important to realize that this algorithm stores values of the pseudoflow $f$ in two different ways. If $\{v, w\}$ is an edge that is not a dynamic tree edge, $f(v, w)$ is stored explicitly, with $\{v, w\}$. If $\{v, w\}$ is a dynamic tree edge, with $w$ the parent of $v$, then $g(v) = u(v, w) - f(v, w)$ is stored implicitly in the dynamic

tree data structure. Whenever a tree edge $(v, w)$ is cut, $g(v)$ must be computed and $f(v, w)$ updated to its current value. In addition, when the algorithm terminates, pseudoflow values must be computed for all edges remaining in dynamic trees.

Two observations imply that the dynamic tree subroutine is correct. First, any edge $\{v, w\}$ that is in a dynamic tree is admissible. By Lemma 2.6.5, in case (1) of *tree-push/update*, vertices $v$ and $w$ are in different trees, and the algorithm never attempts to link a dynamic tree to itself. Second, a vertex $v$ that is not a tree root can have positive balance only in the middle of case (1) of a *tree-push/update* operation. To see this, note that only in this case does the algorithm add flow to a nonroot vertex, and this addition of flow is followed by a send operation that moves the nonroot excess of flow to one or more roots.

**Theorem 2.8.1** *The dynamic tree implementation of Improve-Approximation subroutine runs in $O(nm \log n)$ time.*

*Proof*: First we bound the number of *link* and *cut* operations. The number of *link* operations is at most $7nm$ by a proof like that of Lemma 2.6.4. The number of *cut* operations is at most the number of *link* operations. The total number of *link* and *cut* operations is $O(nm)$. We charge $O(1)$ dynamic tree operations plus $O(1)$ additional work to each *cut* and to each *link* operation. Since each dynamic tree operation requires $O(\log n)$ work, the total amount of work charged to these operations is $O(nm \log n)$.

Consider a *send* operation. Work done at each iteration of the inner loop of *send* is charged to the *cut* operation performed at this iteration. If during an execution of the outer loop of *send* the inner loop is entered, the work done at this outer loop iteration (excluding the work done inside the inner loop which is already accounted for) is charged to the first *cut* operation performed by the inner loop at this iteration of the outer loop. If during an execution of the outer loop the inner loop is not entered, the *send* operation terminates after this iteration of the outer loop. We charge the work done by this iteration to the *link* operation performed in step (1) immediately before the *send* operation has been applied. This accounts for all the work done by *send* operations.

We account for the remaining work as follows. The work done at each execution of step (1) (except for the work done by *send* operations that is already accounted for) is charged to the *link* operation performed at this step. Each occurrence of step (2a) performs a constant amount of work, and the number of such occurrences is $O(nm)$ by a proof like that in Theorem 2.7.2. The work done by each pair of *cut* and *change-value* operations at step (2b) is charged to the *cut* operation. The work done by price updating operations adds up to $O(nm)$ by a proof like that in Theorem 2.7.2. The remaining work done at steps (2b) is $O(1)$ for each price updating operation, for a total of $O(n^2)$.

The total amount of work charged to the *cut* and *link* operations is $O(nm \log n)$. Other work amounts to $O(nm)$ for step (2a) and $O(nm) + O(n^2)$ for step (2b). Running time of the subroutine is therefore $O(nm \log n)$. ∎

**Corollary 2.8.2** *An implementation of the minimum-cost flow algorithm that uses the dynamic tree version of the Improve-Approximation subroutine runs in* $O(nm \log(n) \log(nC))$ *time.*

*Proof*: Immediate from Theorems 2.4.1 and 2.8.1. ∎

## 2.9 Blocking Improve-Approximation Subroutine

In this section we present an alternative approach to the design of the *Improve-Approximation* subroutine. This approach is a generalization of the approach to the maximum flow problem due to Dinic and uses the concept of a blocking flow. The results presented in this section represent joint work with Robert Tarjan, who observed that the blocking flow approach can be extended to obtain an efficient implementations of the *Improve-Approximation* subroutine.

A blocking flow is defined for flow networks (defined in Section 1.2), rather than for circulation networks. Recall that in flow networks all lower bounds on capacity are zero and there are two distinguished vertices, a source $s$ and a sink $t$. A *blocking*

*flow* is a flow with no forward augmenting path, i.e. a flow for which for every path from $s$ to $t$ in the network contains a saturated edge. We use blocking flows only in the context of layered and acyclic networks. A network is *acyclic* if the underlying directed graph is acyclic. An equivalent definition is to say that a network is acyclic if its vertices can be labeled by integers in such a way that for every edge $(v, w)$, we have $label(v) > label(w)$. A network is *layered* [14] if its vertices can be labeled by integers in such a way that for every edge $(v, w)$, we have $label(v) = label(w) + 1$.

There are many algorithms for finding a blocking flow in a layered network. Although a layered network is a special case of an acyclic network, most of these algorithms work in the more general case as well, achieving the same complexity bounds. In particular, the algorithms described in [47,56,72] can be used to find a blocking flow in an acyclic network in $O(n^2)$ time, and the algorithm described in [67] can be used to find a blocking flow in $O(m \log n)$ time.

With a minor modification, the Shiloach-Vishkin algorithm [65] can be used to find blocking flows in acyclic networks in $O(n \log n)$ PRAM time using $O(n)$ processors and $O(n^2)$ memory. A modification is necessary because, unlike a layered network, a general acyclic network does not have even/odd layers. One way to deal with this problem is to introduce a new vertex $i_{(v,w)}$ for every edge $(v, w)$, and replace the edge by two edges, $(v, i_{(v,w)})$ and $(i_{(v,w)}, w)$, both with the same capacity as the edge $(v, w)$. The modified problem is clearly equivalent to the original problem, and in the modified network, the even/odd layers are well defined. The second way to deal with the problem is to modify the algorithm so that the flow pushed into a vertex during a parallel pulse is not processed until the end of the pulse. Both modifications are minor and do not affect the analysis.

Galil's algorithm [27] finds blocking flows in layered networks in $O(n^{2/3}m^{2/3})$ time. This algorithm can also be modified to work on acyclic networks within the same time bound [28].

In this section, we define the notions of an *admissible edge* and an *admissible graph* in a slightly different way than in section 2.6. Given a pseudoflow $f$ and

**Procedure** *Improve-Approximation*$(f, p, \epsilon)$;

```
⟨⟨ initialization⟩⟩
∀(v, w) ∈ V × V do begin
        if c_p(v, w) > 0 then f(v, w) ← l(v, w);
        if c_p(v, w) < 0 then f(v, w) ← u(v, w);
end;
while f is not a circulation do block(f, p, ε);
return(f, p);
```

**end.**

Figure 2.11: The blocking flow Improve-Approximation subroutine.

a price function $p$, we say that an edge $(v, w)$ is admissible if $(v, w) \in E_f$ and $c_p(v, w) < 0$. For a given price function $p$, the admissible graph $G_A$ is the graph of all admissible edges: $G_A = (V, E_A)$, where $E_A = \{(v, w) \in E_f | c_p(v, w) < 0\}$. These definitions are more natural in the context of blocking flows. As we have noticed at the end of Section 2.6, this definition can also be used in the previous sections as well.

We show how to implement the *Improve-Approximation* subroutine using at most $3n$ blocking flow computations. The blocking flow variant of the subroutine, summarized in Figure 2.11, starts by bringing all edges in kilter. Then the subroutine repetively performs the *block* operation described in Figure 2.12 until $f$ is a circulation.

The *block* operation consists of two steps. In the first step, the vertices are partitioned into two sets, $S$ and $\overline{S}$. The set $S$ contains the vertices reachable from some vertex with positive balance in the admissible graph $G_A$, and the set $\overline{S}$ contains all remaining vertices. Then prices of vertices in $S$ are increased by $\epsilon/2$. The second stage constructs an acyclic auxiliary network, finds a blocking flow $f'$ in the auxiliary network, and augments the pseudoflow $f$ by $f'$ (i.e. $f(v, w) \leftarrow f(v, w) + f'(v, w)$ for each $(v, w) \in V \times V$). The auxiliary network $G_{aux} = (V \cup \{s, t\}, E_{aux})$ is constructed

**Procedure** *block*$(f, p, \epsilon)$;

$\langle\langle$ **step 1**$\rangle\rangle$
$S \leftarrow \{w | \exists v$ such that $b(v) > 0$ and $w$ is reachable from $v$ in $G_A\}$;
$\overline{S} \leftarrow V - S$;
$\forall v \in S$ **do** $p(v) \leftarrow p(v) + \epsilon/2$.
$\langle\langle$ **step 2**$\rangle\rangle$
Construct auxiliary network $G_{aux} = (V \bigcup \{s, t\}, E_{aux})$.
Find a blocking flow $f'$ in $G_{aux}$.
Augment $f$ by $f'$.
**return**$(f, p)$;

**end.**

Figure 2.12: The block operation. The running time of the operation depends on the blocking flow subroutine used and on the details of implementation.

from the admissible graph $G_A$ by adding a source $s$, a sink $t$, and edges $(s, v)$ of capacity $b_f(v)$ for every vertex $v \in V$ with $b_f(v) > 0$ and edges $(w, t)$ of capacity $-b_f(w)$ for every vertex $w \in V$ with $b_f(w) < 0$. The capacities of edges $(v, w) \in E_A$ are defined to be $r_f(v, w)$. Note that augmenting $f$ by a flow $f'$ in the auxiliary network results in a valid pseudoflow.

Correctness of the blocking flow subroutine follows from the following lemma.

**Lemma 2.9.1** *At the beginning and at the end of each execution of* block, *pseudoflow $f$ is $(\epsilon/2)$-optimal with respect to $p$, $S \supseteq \{v | b_f(v) > 0\}$, and $\overline{S} \supseteq \{v | b_f(v) < 0\}$. At the beginning and at the end of steps (1) and (2), the admissible graph $G_A$ is acyclic.*

*Proof*: We prove the lemma by induction on the number of *block* operations performed. The basis follows from the fact that after the initialization, we have $E_A = \emptyset$. Now suppose that at the beginning of an execution of *block* the claim of the lemma holds. First we show that the $(\epsilon/2)$-optimality of $f$ is preserved. Note that before step (1) is executed, the reduced cost $c_p(v, w) \geq 0$ for every residual edge

$(v, w) \in S \times \overline{S}$. Since the price change performed during step (1) decreases the reduced cost of such residual edges by $\epsilon/2$, the $(\epsilon/2)$-optimality constraints at these edges remain valid. Consider $(v, w) \in \overline{S} \times S$ such that $(v, w) \in G_f$. If $(w, v) \in G_f$, then the constraints for $(w, v)$ remain valid by the previous case and the constraints for $(v, w)$ remain valid by Lemma 2.3.3. If $(w, v) \notin G_f$, then $f(v, w) = l(v, w)$ and the increase in the reduced cost $c_p(v, w)$ caused by the price change cannot invalidate the corresponding constraint. The reduced costs of the other residual edges do not change. The above observations imply that $f$ is optimal after step (1). Step (2) changes the values of $f$ on admissible edges only, and therefore $f$ remains $(\epsilon/2)$-optimal after step (2).

Suppose that at the beginning of step (1) the admissible graph is acyclic. The price increases performed at step (1) can introduce admissible edges going from $S$ to $\overline{S}$, but these price increases delete all admissible edges going from $\overline{S}$ to $S$. The admissible edges not going across the cut are not affected, and therefore $G_A$ remains acyclic. Step (2) can delete admissible edges but cannot introduce new admissible edges, and therefore $G_A$ remains acyclic after the step.

Finally, suppose that after step (2) there is a path of admissible edges from a vertex with positive balance to a vertex with negative balance. Then there is a forward augmenting path with respect to $f'$ in the auxiliary network, contradicting the fact that $f'$ is a blocking flow. ∎

The following lemma is a generalization of the lemma bounding the number of phases in a layered network algorithm for the maximum flow problem. This lemma is also similar to the lemma that bounds the number of maximum flow computations in a scaling step of a Bland-Jensen algorithm [8], and to Lemma 2.6.2.

**Lemma 2.9.2** *The number of blocking flow computations in an execution of the blocking flow Improve-Approximation subroutine is at most* $3n$.

*Proof*: Note that the balance of a vertex can become positive only during the initialization step of the subroutine and the subroutine terminates when there are

no vertices with a positive balance. Thus there must be a vertex $v$ whose balance is positive during each execution of step (1) of phase operation. By Lemma 2.9.1, the price of $v$ increases by $\epsilon/2$ during such a step, and does not change otherwise. By a similar argument, prices of vertices with a negative balance do not change. By an argument similar to the proof of Lemma 2.6.2, the price of $v$ cannot increase by more then $(3/2)\epsilon$. Therefore the number of phases is at most $3n$. ∎

**Theorem 2.9.3** *Let $B(n,m)$ be the running time of an algorithm that finds a blocking flow in acyclic networks. Then the minimum-cost flow algorithm runs in $O(nB(n,m)\log(nC))$ time.*

*Proof*: Immediate from Theorem 2.4.1 and Lemma 2.9.2. ∎

**Corollary 2.9.4** *The minimum-cost flow problem can be solved in $O(\min(nm\log(n)\log(nC), n^{5/3}m^{2/3}\log(nC), n^3\log(nC)))$ time.*

*Proof*: The lemma follows from Theorem 2.9.3 and on generalizations of blocking flow algorithms [47,27,67] to acyclic networks discussed above. ∎

## 2.10 Parallel and Distributed Implementations

In this section we discuss parallel and distributed implementations of the minimum-cost flow algorithms described earlier. Related work on parallel and distributed algorithms for network flow problems includes a paper of Shiloach and Vishkin [65], and a paper of Bertsekas [6]. Shiloach and Vishkin give an $O(n^2\log n)$ time parallel algorithm for the maximum flow problem, which can be combined with the cost scaling algorithms [62,8] to obtain an $O(n^3\log(n)(\log C))$ time parallel algorithm for the minimum-cost flow problem. Bertsekas [6] exhibits a "chaotic" algorithm for the minimum-cost flow problem that converges in a finite number of steps in a distributed model.

First we discuss two synchronous algorithms. One algorithm gives the better time bound, and the other gives the better memory bound. (In fact, if Conjecture 1 is true, the time bounds for both algorithms are the same to within a constant factor, and the second algorithm is superior). The first algorithm, which we call the *blocking algorithm*, is obtained using the parallel version of the Shiloach-Vishkin blocking flow algorithm to implement the *Improve-Approximation* subroutine as described in Section 2.9. The second algorithm, which we call the *FIFO algorithm*, is obtained by using a parallel implementation of the first-in, first-out version of the *Improve-Approximation* subroutine. This parallel implementation can be obtained from the sequential implementation described in Section 2.7 in the same way as for the first-in, first-out maximum flow algorithm described in Section 1.4.

First we state the performance of parallel versions of the algorithms. The bounds presented below hold in both PRAM [21] and DRAM [53] models of parallel computation.

**Theorem 2.10.1** *The blocking algorithm runs in $O(n^2(\log n)\log(nC))$ parallel time using $O(n)$ processors and $O(n^2)$ memory.*

*Proof*: Immediate from Theorem 2.4.1 and [65]. ∎

**Theorem 2.10.2** *The FIFO algorithm runs in $O(n^3(\log n)(\log nC))$ parallel time using $O(n)$ processors and $O(m)$ memory.*

*Proof*: Similar to the analysis of the parallel maximum flow algorithm (see Section 1.6) using the $O(n^3)$ bound on the number of pulses of the *Improve-Approximation* subroutine instead if the $O(n^2)$ bound which holds in the maximum flow algorithm. ∎

If Conjecture 1 holds, the running time bound for the FIFO algorithm improves by a factor of $n$, and the algorithm becomes superior to the blocking algorithm.

The following two theorems give the performance of the two algorithms in the

synchronous distributed model of parallel computation [31,2]. In the statement of these theorems, $\Delta_p$ denotes the degree of processor $p$ in the network.

**Theorem 2.10.3** *In the synchronous distributed model, the blocking algorithm runs in $O(n^2(\log nC))$ time using $O(n\Delta_p)$ memory per processor $p$ and $O(n^3 \log(nC))$ messages.*

*Proof*: Immediate from Theorem 2.4.1 and [65]. ∎

**Theorem 2.10.4** *In the synchronous distributed model, the FIFO algorithm runs in $O(n^3(\log nC))$ time using $O(\Delta_p)$ memory per processor $p$ and $O(n^2 m \log(nC))$ messages.*

*Proof*: Similar to the analysis of synchronous distributed version of the maximum flow algorithm. ∎

If Conjecture 1 holds then the time complexity bound for the FIFO algorithm improves by a factor of $n$ and the message complexity bound improves by a factor of $m/n$.

Next we discuss implementations of the minimum-cost flow algorithm in the asynchronous distributed model of parallel computation [31,2]. One approach is to use the blocking or FIFO algorithm with the synchronization protocol of [2]. The resulting bounds on message and memory complexity of the algorithm are the same as in Theorems 2.10.3 and 2.10.4, and the resulting time bounds are greater then the bounds given by these theorems by a factor of $\log n$.

Another way to obtain an asynchronous implementation of the algorithm is to use an acknowledgment version of the first-in, first-out algorithm constructed as in the corresponding version of the maximum flow algorithm. The performance of the resulting algorithm is stated in the following theorem. If Conjecture 1 holds, the running time bound given by the theorem improves by a factor of $n$.

**Theorem 2.10.5** *An asynchronous distributed version of the minimum-cost flow algorithm that uses the acknowledgment implementation of the Improve-Approxi-*

*mation subroutine runs in $O(n^3 \log(nC))$ time using $O(\Delta_p)$ memory per processor $p$ and $O(n^2 m \log(nC))$ messages.*

*Proof*: Theorem 2.4.1, Lemma 2.7.3, and the proof of Theorem 1.6.3 are sufficient to prove the theorem if we can show how to detect termination of the *Improve-Approximation* subroutine. To detect termination, we use techniques from [31] and [2]; we assume familiarity with [31,2] in the following discussion.

To detect the termination of the subroutine, we use the fact that the subroutine stops as soon as all vertices that had negative balance at the beginning of the execution have nonnegative balance. At the beginning of the algorithm, we construct a spanning tree for the network. The root of the spanning tree plays a special role in the algorithm (without loss of generality, we assume that the network is connected). At the beginning of each execution of the *Improve-Approximation* subroutine, each processor records the sign of its initial balance, and sends the sign to the root of the tree. The root counts the number of vertices that had negative balance initially. When a balance of a vertex changes from negative to nonnegative, the vertex informs the root about this change, and the root updates its count of the number of vertices with negative balance. Consider the point when every processor has informed the root about the sign of its initial balance, and every vertex whose balance was negative initially has reported the change in sign. At this point, the root knows that the algorithm has terminated, so it broadcasts the termination massage. The number of messages related to termination detection is $O(n^2)$ for an execution of the subroutine, and the additional time is $O(n^2)$. The additional memory required is $O(1)$ for all vertices including the root. ∎

## 2.11 Remarks

The concluding remarks concern practicality and potential improvements and extensions of the minimum-cost flow algorithm.

Initial practical experience with the scaling algorithms for the minimum-cost flow problem turned out to be disappointing. As a result, the algorithms were con-

demned as impractical in the literature, and the simplex-based algorithms continued to be used to solve the minimum-cost flow problem in practice. This situation did not change even though the scaling algorithms were refined and new scaling algorithms were developed for the problem. Recently, however, Bland and Jensen conducted an extensive study [8] to compare an implementation of their scaling algorithm with simplex-based codes. They found their algorithm competitive, and concluded that the practicality of the scaling algorithms should be investigated further. These experimental results agree with the results of Bateson [3] who compared Gabow's scaling algorithm for general matching [26] with the Hungarian method [50].

We believe that our approach yields highly practical algorithms. Our belief is based on the work of Bland and Jensen mentioned above, as well as on the experience with implementations (both sequential and parallel) of the maximum flow algorithm described in Chapter 3 (also see [58]). The experience of Bertsekas and Tseng with an implementation of their algorithm [7] also supports our belief. Further experimental study is needed, however, before the practicality claim can be made with certainty.

The approach presented in this chapter allows a great degree of flexibility. For example, the *Improve-Approximation* subroutine reduces the error parameter of approximation by a factor of 2. The subroutine can be modified to reduce the parameter by a different factor. Although fine-tuning of this factor does not seem to improve the asymptotic running time of the algorithm, it does change the constant factors of the running time and therefore the practical performance of the algorithm. Also, as in the case of the maximum flow algorithm, a different ordering of the basic operations in the generic *Improve-Approximation* subroutine may result in a better performance. As we have mentioned in Section 2.7, we believe that the implementations of the *Improve-Approximation* subroutine that correspond to the first-in, first-out maximum flow algorithm and to the dynamic tree maximum flow algorithm achieve asymptotic running times equivalent to the corresponding maximum flow bounds.

The algorithm can start with any initial price function such that the absolute

value of the difference between the prices of any pair of adjacent vertices is at most $C$ (or even $O(n^k C)$ for a constant $k$). In fact, in a practical implementation the initial price function should be obtained by using a shortest path subroutine. *Improve-Approximation* can also use the shortest path subroutine to update the price function at intermediate points of the execution. This use of the shortest path subroutine is similar to the use of breadth-first search in the maximum flow algorithm discussed in Section 1.7. Another heuristic improvement is to set $\epsilon$ to the largest amount of violation of the complementary slackness conditions before each call to *Improve-Approximation*.

Our algorithm works essentially by scaling costs. It would be interesting to see if a similar approach could be made to work by scaling capacities, or by scaling costs and capacities simultaneously. A possible approach is to order operations of the generic subroutine depending on the change in value (i.e. cost-capacity product) caused by these operations.

Both the maximum flow and the shortest path problems are special cases of the minimum-cost flow problem. Our result implies that the complexity of the minimum-cost flow problem is not too far from the sum of the best upper bounds known for the two subproblems. An interesting extension would be to show how to solve the minimum-cost flow problem in $O(\log(nC))$ applications of any ("black-box") maximum flow and shortest path subroutines. Such a result would imply that the minimum-cost flow problem is not much harder then these two subproblems.

Another interesting open question is the existence of a strongly polynomial $O(n^3 \log^k n)$ (or $O(nm \log^k n)$, if one is able to take advantage of sparsity) algorithm for the minimum-cost flow problem. A potential approach would combine the techniques of [70,59,23,30] with the techniques described in this chapter.

# Chapter 3

# Implementing Parallel Algorithms

## 3.1 Introduction

This chapter describes an implementation of the parallel maximum flow algorithm of section 1.6 on a Connection Machine[1] parallel supercomputer and presents experimental results obtained using the implementation. The implementation makes an extensive use of the parallel prefix operations discussed in the next section. The use of parallel prefix operations was motivated by the work of Blelloch [9]. The material presented in this chapter represents joint research with Charles Leiserson. Most of the work, including all the experimental work, was done while the author was at Thinking Machines, Inc.

In the previous two chapters we have discussed parallel and distributed versions of the network flow algorithms. In this chapter we shall see how practical one of these algorithms is and how it compares with the underlying sequential algorithm. These experimental results are especially interesting because, until recently, large parallel machines were not available commercially, and practical experience with these machines is rare. Although important measures of parallel complexity (running time, processor requirements, communication, and memory) have been identified, the tradeoffs among these complexity measures are not yet understood completely, especially from the practical standpoint. As a result, analysis of a paral-

[1]Connection Machine is a trademark of Thinking Machines, Inc.

lel algorithm does not allow us to predict the practical performance of the algorithm as precisely as does the analysis of a sequential algorithm.

Although implementation of parallel algorithms is an extensive area of research, the scarcity of practical experience with large scale parallel computers makes the choice among several possible implementations difficult. Another difficulty in programming parallel algorithms comes from incomplete understanding of the fundamental parallel operations, i.e., operations used to express algorithms at a high level of abstraction that can be implemented efficiently on a wide class of parallel machines.

We give an abstract model of the Connection Machine architecture that enables us to explain and justify the details of the implementation of the maximum flow algorithm. A detailed description of the design and the capabilities of the machine appears in the book of Hillis [41].

The Connection Machine can be viewed as a distributed memory parallel computer [53]. The machine consists of a large number of processors connected by a routing network. (The biggest machine available now has 64K processors, where $K = 1024$.) Each processor has a local memory. The local memory of a processor can be accessed by other processors via the routing network. The machine is a *single-instruction, multiple-data machine* [18]: the program is stored in a host computer which executes a sequential program containing parallel instructions. When a parallel instruction appears in the program, the host broadcasts it to all processors. Each processor, depending on its memory contents, either executes the instruction or remains idle. The operation of the machine is totally synchronous: the next instruction does not start until all processors have completed the execution of the current instruction.

Due to its distributed nature, the memory organization of the machine is different from the shared memory organization used by some other parallel computers like the Ultracomputer [40]. In the Connection Machine, the interprocessor communication is achieved by allowing a processor to access memory of any other processor using the network. Basic understanding of the memory system of the machine, in

particular the notions of a routing cycle and locality, is essential for the subsequent discussion. The *locality* of memory access is due to the fact that the time required by a processor to access its own memory is much less than the time required by a processor to access memory of another processor. More generally, the time required by a processor to access memory of a processor that is close (in the network) is greater than the time required to access memory of a processor that is far. This brings us to the notion of a routing cycle. Suppose that during the execution of an instruction several processors access memory of other processors. Since the operation of the machine is synchronous, the longest memory access time determines the execution time of the instruction. A *routing cycle* is the amount of time it takes to execute such an instruction. It is important to realize that the routing cycle time varies depending on the interprocessor communication pattern and the size of data being accessed.

When programming any computer at the user level, one wants to abstract the low-level details of the machine architecture. In particular, when programming a parallel computer, one wants to abstract the details of underlying routing network and routing algorithm. How is it possible to do so and still make use of locality? One answer is to use fundamental parallel operations like the parallel prefix operations discussed in the next section.

## 3.2 Parallel Prefix Operations

Parallel prefix operations have been recognized as fundamental parallel operations, and their implementation and use in describing parallel algorithms has been widely studied [9,11,21,49,51,53]. Given an associative binary operation "$*$" and a sequence of $s$ numbers $a_1, a_2, \ldots a_s$, the parallel prefix "$*$" operation maps this sequence into the sequence of $s$ numbers $a_1, a_1 * a_2, \ldots, a_1 * a_2 * \ldots * a_s$.

We are especially interested in the following three operations. The first one is the *prefix-add* operation, obtained by using the plus function. This operation is illustrated in Figure 3.1a. The second operation is the *prefix-copy* operation, obtained by using the projection function $\pi(x, y) = x$. An application of this

(a) *prefix-add*[1 1 1 1 1] = [1 2 3 4 5]

(b) *prefix-copy*[1 2 3 4 5] = [1 1 1 1 1]

(c) *prefix-min*[4 5 2 1 3] = [4 4 2 1 1]

Figure 3.1: Parallel prefix operations.

(a) *seg-prefix-add*([1 1 1][1 1][1][1 1 1]) = ([1 2 3][1 2][1][1 2 3])

(b) *seg-prefix-copy*([1 2 3][4 5][6][7 8 9]) = ([1 1 1][4 4][6][7 7 7])

(c) *seg-prefix-min*([3 2 1][4 5][6][7 9 8]) = ([3 2 1][4 4][6][7 7 7])

Figure 3.2: Segmented parallel prefix operations.

operation to a list of numbers results in copying the value of the first element of the list to all other elements of the list, as illustrated in Figure 3.1b. The third operation, *prefix-min*, is obtained using the minimum function. This operation is illustrated in Figure 3.1c. After this operation is applied, the last element of the list is equal to the smallest element of the input list.

For our implementation we need an extension of parallel prefix operations. Given an associative binary operation "$*$", a *segmented parallel prefix operation* maps a list of sequences $S_1, S_2, \ldots, S_l$ into the list $S'_1, S'_2, \ldots, S'_l$ where each sequence $S'_i$ is obtained from the corresponding sequence $S_i$ using the parallel prefix "$*$" operation. The lengths of the input sequences $S_i$ can be different. Segmented parallel prefix operations corresponding to addition, projection, and minimum functions are illustrated in Figure 3.2.

*Parallel suffix operations* are defined similar to the parallel prefix operations, but enumerating sequences in reverse order. Segmented parallel suffix operations are illustrated in Figure 3.3.

The parallel prefix operations, segmented or not, can be implemented efficiently on any distributed memory parallel machine such that a balanced binary tree can

(a) *seg-suffix-add*([1 1 1][1 1][1][1 1 1]) = ([3 2 1][2 1][1][3 2 1])

(b) *seg-suffix-copy*([1 2 3][4 5][6][7 8 9]) = ([3 3 3][5 5][6][9 9 9])

(c) *seg-suffix-min*([3 2 1][4 5][6][7 9 8]) = ([1 1 1][4 5][6][7 8 8])

Figure 3.3: Segmented parallel suffix operations.

be embedded in the routing network of the machine. In particular, parallel prefix operations have been implemented on the Connection Machine [9]. To understand the use of these operations on the Connection Machine, one has to know that the processors on the machine are numbered. A sequence of numbers is represented in the machine by placing elements of the sequence in consequent processors. The processors containing the first and the last elements of the list are marked appropriately.

The efficiency of a parallel prefix computation depends of complexity of the binary operation "$*$" and on the size of data items. For simple operations like addition, projection, and minimum, the time required to perform parallel prefix operations is of the same order of magnitude as the time required for one routing cycle on data of the same size. Although the routing cycle time and time to perform a parallel prefix operation varies depending on the communication pattern and on the type of the parallel prefix operation, we shall refer to these times as to the routing cycle in our description of the implementation, since this gives a good enough intuitive measure of performance. The important point to remember is that the routing cycle is much greater then the local instruction execution time.

## 3.3 Implementation Details

We start the description of the implementation by describing the mapping of the input network into the machine. This mapping is due to Blelloch [9] who used it to implement several graph algorithms on the Connection Machine.

In our description of the mapping, it is important to realize that processors of the machine are linearly ordered in a way that allows efficient implementation of the segmented parallel prefix and suffix operations described in the previous section. Each vertex $v$ is assigned a processor $P_v$. Each undirected edge $\{v, w\}$ is assigned two processors, $P_{(v,w)}$ and $P_{(w,v)}$. We call processors $P_{(v,w)}$ and $P_{(w,v)}$ *pair processors*. Vertex and edge processors are selected so that a vertex processor $P_v$ is immediately followed by the processors $P_{(v,w)}$ that correspond to edges incident to $v$, and the last edge on the list is marked as such. This organization allows us to perform segmented parallel prefix and suffix operations on sequences consisting of vertices with their edges. It is important to realize that the information associated with an undirected edge $\{v, w\}$ is stored in both $P_{(v,w)}$ and $P_{(w,v)}$. In addition to the information associated with the edge $\{v, w\}$. processor $P_{(v,w)}$ also stores the address of its pair processor $P_{(w,v)}$, and the distance label values $d(v)$ and $d(w)$.

The initialization step of the algorithm is easy to implement, so we shall concentrate on implementation of the main loop (see section 1.6). The main loop consists of application of the *pulse* procedure until these are no active vertices. The *pulse* procedure is the only nontrivial part of the implementation. The implementation of this procedure is summarized on Figure 3.4.

Steps (1)-(3) implement the first stage of the *pulse* procedure. Step 1 copies the value of the excess at each vertex processor $P_v$, $v \in V - \{s, t\}$, to the edge processors $P_{(v,w)}$ using a *seg-prefix-copy* operation. The main purpose of step (2) is to distribute vertex excesses among outgoing edges. Our implementation favors edges at the end of edge lists. For each edge $(v, w)$, we first compute the maximum amount that can be pushed through the edge. This amount is equal to $c_f(v, w)$ if $d(v) = d(w) + 1$ and to zero otherwise. Then, a *seg-suffix-add* operation is executed on these values. Now, each edge processor $P_{(v,w)}$ has information about the excess to be pushed from $v$, the amount it can push, and the amount that can be pushed through the edges that follow $(v, w)$ on the edge list of $v$. This information is enough to compute the amount $\sigma(v, w)$ to be pushed through the edge $(v, w)$. After the execution of the *seg-suffix-add* operation, each vertex processor $P_v$ contains the information about

**Procedure** *Pulse.*

**(1)** **For all** $v \in V - \{s,t\}$ *copy* $e(v)$ *to all* $P_{(v,w)}$ *using seg-prefix-copy operation*

**(2)** ⟨⟨ distribute excesses⟩⟩

**For all** $P_{(v,w)}$ compute, using *seq-suffix-add*, the amount that can be pushed to lower-labeled vertices through edges that follow $(v,w)$ on the edge list of $v$.

**For all** $P_{(v,w)}$ compute the amount $\sigma(v,w)$ to be pushed from $v$ to $w$.

**For all** $v \in V$ compute the amount of excess that remains at $v$ after the pushing.

**(3)** ⟨⟨ push flow⟩⟩

**For all** $P_{(v,w)}$ **do if** $\sigma(v,w) > 0$ **then begin**

$f(v,w) \leftarrow f(v,w) + \sigma(v,w)$; $c_f(v,w) \leftarrow c_f(v,w) - \sigma(v,w)$;

send a message containing $\sigma(v,w)$ to $P_{(w,v)}$;

**end.**

**For all** $P_{(w,v)}$ that received message $\sigma(v,w)$ **do begin**

$f(w,v) \leftarrow f(w,v) - \sigma(v,w)$; $c_f(w,v) \leftarrow c_f(w,v) + \sigma(v,w)$;

**end.**

**(4)** ⟨⟨ compute new distance labels⟩⟩

**For all** $P_{(v,w)}$ **do**

**if** $c_f(v,w) > 0$ **then** *head-label*$(v,w) \leftarrow d(w)$

**else** *head-label*$(v,w) \leftarrow 2n$.

**For all** $v \in V - \{s,t\}$ compute *new-d*$(v)$ using *seg-suffix-min.*

**(5)** **For all** $v \in V - \{s,t\}$ copy *new-d*$(v)$ to all $P(v,w)$ using *seg-prefix-copy.*

**(6)** ⟨⟨ broadcast changed labels⟩⟩

**For all** $P_{(v,w)}$ such that $v \notin \{s,t\}$ **do**

**if** $d(v) \neq new - d(v)$ **then**

set the value of $d(v)$ at $P_{(w,v)}$ to *new-d*$(v)$.

**(7)** ⟨⟨ update excesses⟩⟩

**For all** $w \in V$ **do begin**

compute, using *seg-suffix-add*, the amount of flow *new-e*$(w)$ pushed into $w$;

$e(w) \leftarrow e(w) + new - e(w)$;

**end.**

**end.**

Figure 3.4: Implementation of the pulse procedure.

how much excess can be pushed from the vertex $v$ at this pulse. The processor sets the value of its variable $e(v)$ to the amount that will remain after the pushing.

Finally, in step (3), all edge processors $P_{(v,w)}$ for which the amount $\sigma(v,w)$ computed in step (2) is positive, increase $f(v,w)$ by $\sigma(v,w)$, decrease $c_f(v,w)$ by the same amount, and send a message containing $\sigma(v,w)$ to their dual processors $P_{(w,v)}$ (i.e., writes to the agreed-upon location in the memory of $P_{(w,v)}$). Each processor $P_{(w,v)}$ that receives such a message decreases $f(w,v)$ by $\sigma(v,w)$ and increases $c_f(w,v)$ by the same amount.

Steps (4)-(6) implement the second stage of the *pulse* procedure. First, each edge processor $P_{(v,w)}$ sets its variable *head-label* to either the value of $d(w)+1$ if the edge $(v,w)$ is not saturated or to $2n$ if the edge is saturated. Recall that the value of $d(w)$ is stored locally at $P(v,w)$, so the above computation is performed without using the router of the machine. Also, note that the new label being computed is less than $2n$, which explains the use of $2n$ as a value of *head-label* for saturated edges. Next, the *seg-suffix-min* operation is performed on the *head-label* variable, and as a result each vertex processor $P_v$ contains the new value of $d(v)$. In step (5), all vertex processors except $P_s$ and $P_t$ copy this new value to their edge processors using a *seg-prefix-copy* operation. In step (6), each edge processor $P_{(v,w)}$ checks if the new value of $d(v)$ is different from the old value and if it is, the processor updates its value of $d(v)$ and the value of $d(v)$ of its pair processor $P_{(w,v)}$ (via the routing network).

Step (7) implements the third stage of the *pulse* subroutine. The step performs the *seg-suffix-add* operation to compute, for each vertex $v$, the amount of flow pushed into $v$ during step (3). This amount is added to the excess $e(v)$ updated at step (2) to account for the flow pushed out of $v$.

The running time of each step is slightly greater than a routing cycle of the machine, because the running time of each step is dominated by a segmented prefix or suffix operation (steps (1), (2), (4), (5), (7)) or by a message routing operation (steps (3), (6)). The overall running time of the *pulse* procedure is roughly seven

routing cycles of the machine, which results in a tight inner loop: the key factor in the performance of the implementation.

We conclude this section with a discussion of the implementation. This implementation is done in the spirit of data parallelism [42]: each item of data is assigned a processor which operates on this item. This approach is natural for the Connection Machine with its fine-grained parallelism, characterized by a low memory to processor ratio.

What are the tradeoffs of the implementation? The advantages of the implementation are a tight inner loop and a simple but effective use of locality through parallel prefix operations. The disadvantage is a relatively low processor utilization. As we have mentioned in Section 1.6, the algorithm can be implemented with $O(n)$ processors instead of $O(m)$ processors used by the implementation we have described. The alternative implementation achieves better processor utilization. This implementation, however, requires processor scheduling, which introduces extra overhead and makes the use of locality difficult.

The above implementation is designed for the Connection Machine architecture, but it is also good for other kinds of architectures. For example, fetch-and-add operations can be used to implement parallel prefix operations on the Ultracomputer [40].

## 3.4 Experimental Results

We start this section by describing the actual implementation, including the heuristics used to speed up the algorithm. Then we describe the generator of examples used to obtain the input data. Finally, we present and discuss the experimental results.

The actual code implemented the minimum-cut variation of the maximum flow algorithm, as described in Section 1.7 and in [37]. The main difference of the mincut algorithm is that a vertex becomes "dead" as soon as its label reaches or exceeds $n$

(in this case, the sink is not reachable from the vertex). The implementation uses two heuristic improvements. The first improvement is to declare a vertex "dead" as soon as the value of its label exceeds $n$ minus the number of "dead" vertices. The second improvement is to do breadth-first search backwards from the sink in the residual graph to update the distance labels to be the exact distances to the sink. The frequency of the breadth-first search updates is determined by the size of the graph and the number of relabeling operations since the previous breadth-first search update.

The experiments have two goals: determine the practicality of the parallel implementation and compare the parallel implementation with the sequential implementation of the same algorithm. In addition to the parallel implementation on the 32K Connection Machine, the corresponding first-in, first-out algorithm was implemented on the Symbolics 3600 series Lisp Machine. The parallel implementation is written in *LISP and the sequential implementation in LISP. Both implementations use the heuristics described above. The only difference is in the tuning of the breadth-first search update frequency. Different tuning is needed because of the different relative costs of breadth-first search in the parallel and sequential implementations.

Our experiments are conducted on large and difficult examples of the minimum-cut problem. Generating such examples is a non-trivial task. Several methods we tried either generated trivial examples (where either the source or the sink is on one side of the cut and all other vertices are on the other side) or simple examples. On these examples, the programs run extremely fast, and the distribution of operation performed during an execution is different from what the analysis predicts: relabeling operations dominate the sequential running time. The method described below produces examples that take longer time and produce the distribution of operations that agrees with the worst-case analysis, with nonsaturating pushes dominating the sequential running time.

To see how examples are generated, imagine an infinite pipe with a mesh drawn on it. Suppose the pipe goes from west to east. The distance from a vertex of the mesh to the nearest neighbor in both the horizontal and vertical directions is one,

and circumference of the pipe is $D$. First we construct a graph on the vertices of the mesh. In this graph, every vertex will have degree $4\Delta$, were $\Delta$ is assumed to be less then $D/2$. To construct the graph, we connect each vertex $v$ by a directed edge to all vertices $w$ within distance $\Delta$ due east, west, south, and north from $v$. The capacity of an edge $(v, w)$ is defined depending on the distance $x$ between $v$ and $w$. The capacity is selected at random from a uniform distribution on the interval $[0, 2^{-x})$.

To complete the construction, we introduce a source $s$ and a sink $t$. Then we "cut" the pipe by two planes perpendicular to the axis of the pipe. The cutting planes are distance $L$ apart. We disregard the portion of the pipe that is not between the planes. The edges cut by one plane are connected to the source preserving their directions and capacities; the edges cut by the other plane are connected to the sink in the similar manner. Note that since a single vertex may have several of its edges cut by a plane, a single vertex may have several edges connecting it with the source or the sink. These multiple edges are merged together, and the capacity of the resulting edge is set to the sum of capacities of the merged edges.

Why is an average problem constructed by such a generator hard? In the network, short source-to-sink paths have small residual capacity, and long source-to-sink paths have large residual capacity. The algorithm tends to processes short paths first. Saturating short paths does not, however, saturate the cut, so the algorithm must continue.

In our experiments, the values of parameters are selected using the following formulas: $L = D = \lfloor \sqrt{n} \rfloor$, $\Delta = \lfloor (\sqrt{n} - 1)/2 \rfloor$. The value of $\Delta$ determines the graph density and is selected to produce networks of moderate density. The values of $n$ are selected so that $\sqrt{n}$ is integer and the *size* of the problem, i.e., the number of processors required by our implementation, is close to a multiple of the machine size. Two sequences of examples were evaluated. Tables 3.1a and 3.1b summarize the experimental data.

As one can see from the table, the size of some examples is greater than 32K, the size of the Connection Machine used in our experiments. In these cases, the built-in

(a)

| n | Size | Running Time | | Speedup |
|---|---|---|---|---|
| | | Lisp Machine | Connection Machine | |
| 531 | 15,435 | 56.8 | 1.98 | 29 |
| 843 | 31,699 | 195.3 | 3.24 | 60 |
| 1371 | 65,307 | 389.8 | 4.42 | 88 |
| 2211 | 130,803 | 1111.3 | 9.98 | 111 |

(b)

| n | Size | Running Time | | Speedup |
|---|---|---|---|---|
| | | Lisp Machine | Connection Machine | |
| 531 | 15,435 | 81.2 | 2.33 | 35 |
| 843 | 31,699 | 142.7 | 2.93 | 49 |
| 1371 | 65,307 | 303.1 | 4.47 | 68 |
| 2211 | 130,803 | 1734.5 | 9.58 | 181 |

Table 3.1: Experimental results for the first (a) and the second (b) sequence of examples. All running times are in seconds and exclude paging time.

virtual processor mechanism of the Connection Machine is used. This mechanism is simple but effective, and does not introduce any overhead. Our biggest examples require each real processor to serve four virtual processors. The first example in each sequence is much smaller than the machine size of 32K, so the processor utilization is low and the speedup compared to the sequential implementation is low as well.

The running times given in the table are in seconds and do not include paging time, which is a problem for the Lisp Machine on larger examples. The running times for the parallel implementation are quite good: under ten seconds even for the largest examples.

How does the parallel implementation compare with the sequential implementation? The speedup varies from under 30 on small examples (when not all processors are utilized) to well over a hundred on the largest example . In general, the speedup increases with the problem size. Two factors are responsible for this increase. First, bigger problems exhibit larger amounts of parallelism. Second, a larger number of virtual processors per real processor increases the locality of data and therefore decreases the relative cost of parallel prefix operations.

Several simple enhancements in Connection Machine systems software will significantly improve the performance of the parallel implementation. With these enhancements, we expect a 600-700 times speedup for the larger 64K Connection Machine on networks of size around 512K.

In conclusion, we would like to emphasize that our experimental results are highly sensitive to the conditions of the experiment. For example, using higher-degree graphs would probably increase the speedup achieved by the parallel implementation, and using lower-degree graphs would decrease the speedup. In the low-degree case, the Ahuja-Orlin sequential algorithm [1] mentioned in Section 1.1 should produce better results then the sequential algorithm that we have implemented. Since many maximum flow problems that appear in practice are much smaller and much simpler than the examples we have generated, most network flow algorithms produce satisfactory results In Chapter 2 we have seen, however, that minimum-cost flow problems can be solved by iterating a generalized version of the maximum flow algorithm. Since large and hard instances of the minimum-cost flow problem do appear in practice, fast maximum flow algorithms are very important in this context.

# Chapter 4

# Parallel Symmetry-Breaking

## 4.1 Introduction

In the previous chap rs we have seen the need for new methods in design and implementation of efficient parallel algorithms. One way to identify these methods is to study problems for which simple but efficient sequential algorithms exist, but which appear to be much harder to solve efficiently in a parallel framework. A known example of a problem with a trivial sequential algorithm which is hard to solve in parallel is the problem of finding a maximal independent set in a graph [73]. This problem was shown to be in the class NC of problems which can be solved in polylogarithmic time using a polynomial number of processors by Karp and Wigderson [46]. A simple randomized algorithm for the problem is due to Luby [55]. Recently M. Goldberg and Spencer [39] gave a deterministic algorithm for the problem that runs in polylogarithmic time using a linear number of processors.

The study of the maximal independent set problem shows the importance of techniques for breaking symmetry in parallel. The symmetry-breaking comes up in many other parallel algorithms as well. In many cases, however, it is enough to be able to break symmetry in special kinds of graphs. The performance of the resulting algorithm improves if we can solve the special case of symmetry-breaking more efficiently.

In this chapter we present a technique for breaking symmetry. In particular, we give an $O(\lg^* n)$ time algorithm to 3-color a rooted tree. Our technique can be viewed as a generalization of the deterministic coin-flipping technique of Cole and Vishkin [13]. To show the usefulness of our technique, we present the following algorithms. All of the algorithms presented use a linear number of processors.

- For graphs whose maximum degree is a constant $\Delta$, we give an $O(\Delta \log \Delta \lg^* n)$ algorithm for $(\Delta+1)$-coloring and for finding a maximal independent set on an EREW PRAM.
- We give an algorithm to 7-color a planar graph. Both this algorithm and the maximal independent set (for planar graphs) algorithm based on it run in $O(\log n \lg^* n)$ time on a CRCW PRAM and in $O(\log^2 n)$ time on an EREW PRAM. We also give an $O(\log^3 n \lg^* n)$ CRCW algorithm to 5-color a planar graph.
- We give an $O(\log n \lg^* n)$ time algorithm for finding a maximal matching in a planar graph on a CRCW PRAM.

The results stated above improve the running time and processor bounds for the respective problems. The fastest previously known algorithm for $(\Delta+1)$-coloring [55], in the case of constant-degree graphs, runs in $O(\log n)$ time, and the deterministic version of this algorithm requires $n^3$ processors. The 5-coloring algorithm for planar graphs, due to Boyar and Karloff, [10] runs in $O(\log^5 n)$ time, but the deterministic version of this algorithm requires $n^3$ processors. The $O(\log^3 n)$ running time of the maximal matching algorithm due to Israeli and Shiloach [44] can be reduced to $O(\log^2 n)$ in the restricted case of planar graphs, but our algorithm is faster.

Although in this chapter we have limited ourselves to the application of our techniques to the PRAM model of computation, the same techniques apply to a distributed model of computation [31,2]. Moreover, the $O(\lg^* n)$ lower bound given by Linial [54] for the maximal independent set problem on a chain in the distributed model shows that our symmetry-breaking technique is optimal in this model.

The fact that a rooted tree can be 3-colored in $O(\lg^* n)$ time raises the question whether a rooted tree can be 2-colored within the same time complexity. We answer this question by giving an $\Omega(\log n/\operatorname{loglog} n)$ lower bound for 2-coloring a rooted tree. We also present an $\Omega(\log n/\operatorname{loglog} n)$ lower bound for finding a maximal independent set in a general graph, thus answering the question posed by Luby [55].

This chapter represents joint research with Plotkin [34,35]. Results similar to many of the results presented in this chapter were obtained independently by Shannon [63]. A joint paper will appear in [36].

## 4.2 Definitions and Notation

This section describes our assumptions about the computational model and introduces the notation used throughout the chapter. As in the earlier chapters, we use $n$ to denote the number of vertices and $m$ to denote the number of edges in a graph. We use $\Delta$ to denote the maximum degree of the graph.

Given a graph $G = (V, E)$, we say that a subset of vertices $I \in V$ is *independent* if no two vertices in $I$ are adjacent. A *coloring* of a graph $G$ is an assignment $C : V \rightarrow N$ of positive integers (colors) to the vertices of the graph. A coloring is *valid* if no two adjacent vertices have the same color. The $i^{\text{th}}$ bit in the color of a vertex $v$ is denoted by $C_v(i)$. A subset of edges $M \in E$ is a *matching* if any two distinct edges in $M$ have no vertices in common.

The following problems are discussed in this chapter:

- The vertex-coloring (VC) problem: find a valid coloring of a given graph that uses at most $\Delta+1$ colors.
- The maximal independent set (MIS) problem: find a maximal independent set of vertices in a given graph.
- The maximal matching (MM) problem: find a maximal matching in a given graph.

We make a distinction between *unrooted* and *rooted* trees. In a rooted tree, each nonroot vertex knows which of its neighbors is its parent.

The following notation is used:

$$\begin{aligned} \lg x &= \log_2 x \\ \lg^{(1)} x &= \lg x \\ \lg^{(i)} x &= \lg\lg^{(i-1)} x \\ \lg^* x &= \min_{\{i \mid \lg^{(i)} x \leq 2\}}(i) \end{aligned}$$

We assume a PRAM model of computation where each processor is capable of executing simple word and bit operations. The word width is assumed to be $O(\log n)$. The word operations we use include bit-wise boolean operations, integer comparisons, and unary-to-binary conversion. In addition, we assume that each processor has a unique *identification number* $O(\log n)$ bits wide, which we denote by PE-ID. We use the exclusive-read, exclusive-write (EREW) PRAM, the concurrent-read, exclusive-write (CREW) PRAM, or the concurrent-read, concurrent-write (CRCW) PRAM as appropriate. For a discussion of the PRAM model of computation, see [21]. All lower bounds are proved for a CRCW PRAM with a polynomial number of processors.

## 4.3 Coloring Rooted Trees

This section describes an $O(\lg^* n)$ time algorithm for 3-coloring rooted trees. First we describe an $O(\lg^* n)$ time algorithm for 6-coloring rooted trees. Then we show how to transform a 6-coloring of a rooted tree into a 3-coloring in constant time.

The procedure *6-Color-Rooted-Tree* is shown in Figure 4.1. This procedure accepts a rooted tree $T = (V, E)$ and 6-colors it in time $O(\lg^* n)$. Starting from the valid coloring given by the processor ID's, the procedure iteratively reduces the number of bits in the color descriptions by recoloring each nonroot vertex $v$ with the color obtained by concatenating the index of a bit in which $C_v$ differs from $C_{father(v)}$ and the value of this bit. The root $r$ appends $C_r[0]$ to 0.

```
Procedure 6-Color-Rooted-Tree.
    L ← ⌈lg n⌉;
    for all v ∈ V in parallel do C_v ← PE-ID(v);   ⟨⟨ initial coloring⟩⟩
    while L > ⌈lg L + 1⌉ for all v ∈ V in parallel do begin
          if v is the root then begin
            i_v ← 0;
            b_v ← C_v(0);
          end;
          else begin
            i_v ← min_{i | C_v(i)≠C_father(v)(i)}(i);
            b_v ← C_v(i_v);
          end;
          C_v ← b_v i_v;
    end;
end.
```

Figure 4.1: The coloring algorithm for rooted trees

**Theorem 4.3.1** *The algorithm 6-Color-Rooted-Tree produces a valid 6-coloring of a tree in $O(\lg^* n)$ time on a CREW PRAM using $O(n)$ processors.*

*Proof*: First we prove by induction that the coloring computed by the algorithm is valid, and then we prove the upper bound on the execution time.

Assume that the coloring $C$ is valid at the beginning of an iteration. We shall show that the coloring at the end of the iteration is also valid. Let $v$ and $w$ be two adjacent vertices; without loss of generality assume that $v$ is the father of $w$. By the algorithm, $w$ chooses some index $i$ such that $C_v(i) \neq C_w(i)$ and $v$ chooses some index $j$ such that $C_v(j) \neq C_{father(v)}(j)$. The new color of $w$ is $\langle i, C_w(i)\rangle$ and the new color of $v$ is $\langle j, C_v(j)\rangle$. If $i \neq j$, the new colors are different and we are done. On the other hand, if $i = j$, then $C_v(i) \neq C_w(i)$ and again the colors are different. Hence, the validity of the coloring is preserved.

Now we show that the algorithm terminates after $O(\lg^* n)$ iterations. Let $L_k$ denote the number of bits in the representation of colors after $k$ iterations. For

$k = 1$ we have

$$\begin{aligned} L_1 &= \lceil \lg L \rceil + 1 \\ &\leq 2\lceil \lg L \rceil \end{aligned}$$

if $\lceil \lg L \rceil \geq 1$.

Assume for some $k$ we have $L_{k-1} \leq 2\lceil \lg^{(k-1)} L \rceil$ and $\lceil \lg^{(k)} L \rceil \geq 2$. Then

$$\begin{aligned} L_k &= \lceil \lg L_{k-1} \rceil + 1 \\ &\leq \lceil \lg(2 \lg^{(k-1)} L) \rceil + 1 \\ &\leq 2\lceil \lg^{(k)} L \rceil \end{aligned}$$

Therefore, as long as $\lceil \lg^{(k)} L \rceil \geq 2$,

$$L_k \leq 2\lceil \lg^{(k)} L \rceil.$$

Hence, the number of bits in the representation of colors $L_k$ decreases until, after $O(\lg^* n)$ iterations, $\lceil \lg^{(k)} L \rceil$ becomes 1 and $L_k$ reaches the value of 3 (the solution of $L = \lceil \lg L \rceil + 1$). Another iteration of the algorithm produces a 6-coloring: 3 possible values of the index $i_v$ and 2 possible values of the bit $b_v$. The algorithm terminates at this point.

We use the concurrent-read capability to enable all the sons of a vertex $v$ to access the color of $v$ simultaneously; no concurrent-write capabilities are required. For constant-degree trees the concurrent-read capability is not needed. ∎

As we have shown, a rooted tree can be 6-colored quickly. A natural question to ask at this point is whether one can use fewer colors and still stay within the same complexity bounds. The following theorem answers this question.

**Theorem 4.3.2** *A rooted tree can be 3-colored in $O(\lg^* n)$ CREW PRAM time using $O(n)$ processors.*

*Proof*: The algorithm *3-Color-Rooted-Tree* presented in Figure 4.2 starts by using the previously described algorithm to 6-color the tree and then recolors it in 3 colors in constant time.

The algorithm recolors the vertices colored with *bad* colors 3, 4, and 5, into *good* colors 0, 1, 2 as follows. First, each vertex is recolored in the color of its father, so that any two vertices with the same father have the same color. The root, which has no father, recolors itself with a color different from its current color. Next, the algorithm removes the color from every vertex that has a bad color and has a neighbor with a good color. These vertices become uncolored. Every vertex $v$ that still has a color $C_v$ is recolored in the color $C_v \bmod 3$; this gets rid of the remaining bad colors. Note that this coloring has the property that for any vertex $v$, all of the sons of $v$ that are colored must have the same color.

The resulting coloring is valid, but not all vertices are colored. By the construction, every uncolored vertex has at least one colored neighbor. Therefore, if there are two vertices $v$ and $w$, such that $v = father(w)$ and both vertices are uncolored, then $father(v)$ is colored and $sons(w)$ are colored too. The algorithm colors $v$ with a color different from $C_{sons(v)}$ and from $C_{father(v)}$. Such a color always exists because there are 3 different colors to choose from and all the colored sons of $v$ have the same color. Finally, the algorithm colors $w$ with a color different from both $C_v$ and $C_{sons(w)}$. Every step of the *3-Color-Rooted-Tree* algorithm can be executed in constant time except for the first step, in which we color the tree with 6 colors. Hence, the total running time of the algorithm is $O(\lg^* n)$. ∎

*Remark*: Theorem 4.3.2 holds for a more general class of graphs containing all directed graphs such that out-degree of every vertex is at most one. The same proof applies with one small modification. Since the concept of a root makes no sense for graphs which are not trees, the proof should use a more general concept of a vertex with zero out-degree instead.

Any tree can be 2-colored. In fact, it is easy to 2-color a tree in polylogarithmic time. For example, one can use treefix operations [53] to compute the distance from each vertex to the root, and color even level vertices with one color and odd level vertices with the other color. It is harder to find a 2-coloring of a rooted tree in parallel, however, than it is to find a 3-coloring of a rooted tree. In section 4.6 we show a lower bound of $\Omega(\log n / \log\log n)$ on 2-coloring of a directed list by a CRCW PRAM with a polynomial number of processors, which implies the same

**Procedure** *3-Color-Rooted-Tree.*
$C \leftarrow$ *6-Color-Rooted-Tree* $(V, E)$;
**for all** $v \in V$, $v \neq$ root **in parallel do** $C_v \leftarrow C_{father(v)}$;
$C_{root} \leftarrow \min_{i \in \{0,1,2\} - \{C_{sons(root)}\}} (i)$;
$V_1 \leftarrow \{v \mid C_v \leq 2\}$;
$V_2 \leftarrow V - V_1$;
$V' \leftarrow \{v \mid v \in V_2 \text{ and } \exists (v, w) \in E, w \in V_1\}$;
**for all** $v \in V - V'$ **in parallel do** $C_v \leftarrow C_v \bmod 3$;
**for all** $v \in V'$ **in parallel do** $C_v \leftarrow$ uncolored;
**for all** $v \in V'$ **in parallel do**
  **if** father(v) $\notin V'$ **then begin**
    $C_v \leftarrow \min_{i \in \{0,1,2\} - \{C_{sons(v)}\} - \{C_{father(v)}\}} (i)$;
    $V' \leftarrow V' - \{v\}$;
  **end;**
**for all** $v \in V'$ **in parallel do**
  $C_v \leftarrow \min_{i \in \{0,1,2\} - \{C_{sons(v)}\} - \{C_{father(v)}\}} (i)$;
**end.**

Figure 4.2: The 3-coloring algorithm for rooted trees

lower bound for rooted trees.

## 4.4 Coloring Constant-Degree Graphs

The method for coloring rooted trees described in the previous section is a generalization of the deterministic coin-flipping technique described in [13]. The method can be generalized even further [35] to color constant-degree graphs in a constant number of colors. In the generalized algorithm, the current color of a vertex is replaced by a new color obtained by looking at each neighbor, appending the index of a bit in which the current color of the vertex is different from the neighbor's color to the value of the bit in the vertex color, and concatenating the resulting strings. This algorithm runs in $O(\lg^* n)$ time, but the number of colors, although constant, is exponential in the degree of the graph. In this section we show how to use the procedure *3-Color-Rooted-Tree* described in the previous section to color a

```
Procedure Find-Forest(V,E).
    E' ← ∅;
    R ← ∅;
    for all v ∈ V in parallel do ⟨⟨ construct the forest – the first step⟩⟩
        if PE-ID(v) is not a local maximum then begin
          e_v ← (v, w) s.t. (v, w) ∈ E and PE-ID(w) = max{PE-ID(u)|(v, u) ∈ E};
          E' ← E' ∪ e_v;
        end;
        else R ← R ∪ v
    for all v ∈ R in parallel do ⟨⟨ get rid of zero-depth trees – the second step⟩⟩
        if ∄(v, w) ∈ E' and ∃(v, w') ∈ E then
          E' ← E' ∪ (v, w');
    return (E'); ⟨⟨ the edges of the forest⟩⟩
end.
```

Figure 4.3: The spanning forest algorithm

constant-degree graph with $(\Delta+1)$ colors, where $\Delta$ is the maximum degree of the graph, in $O(\Delta^2 \log(\Delta) \lg^* n)$ time.

First, we describe how to find in constant time a forest in a given graph such that each vertex with nonzero degree in the graph has nonzero degree in the forest. The removal of the edges of the forest decreases the maximum degree of the remaining graph (unless the maximum degree of the graph is zero). We shall use this property (later) to decompose the edges into $\Delta$ sets, each set inducing a forest on the vertices of the graph. The procedure *Find-Forest* (see Figure 4.3) constructs such a forest.

The procedure has two steps. In the first step each vertex compares the ID's of its neighbors with its own ID. A vertex that does not have the maximum processor ID among its neighbors chooses an edge that connects it to the neighbor with the largest processor ID. The graph induced by the chosen edges is a forest (this graph has no cycles) and the vertices with the highest processor ID's among their neighbors – local maxima – are roots of the forest. In the second step each root with no sons chooses an edge that connects it to one of its neighbors. The roots are local maxima and are therefore independent. Hence, no new cycles are introduced into the graph

```
Procedure Color-Constant-Degree-Graph.
    E' ← E;
    i ← 0;
    while E' ≠ ∅ do begin  ⟨⟨ the first phase⟩⟩
          E_i ← Find-Forest(V, E');
          E' ← E' − E_i;
          i ← i + 1;
    end;
    for all v ∈ V in parallel do  ⟨⟨ initial coloring⟩⟩
        C(v) ← 1;
    for i ← i − 1 to 0 do begin  ⟨⟨ the second phase⟩⟩
        C' ← 3-Color-Rooted-Tree (V, E_i);
        E' ← E' + E_i;
        for k ← 1 to 3 do
            for j ← 1 to Δ+1 do begin
                V' ← V;
                for all v ∈ V' in parallel do
                    if C(v) = j and C'(v) = k then begin
                      C(v) ← max_{i∈{{1,2,...Δ+1}−{C(w) | (v,w)∈E'}}}(i);
                      V' ← V' − {v};
                    end;
                end;
    end;
end.
```

Figure 4.4: The algorithm for coloring constant-degree graphs

induced by the chosen edges.

The algorithm *Color-Constant-Degree-Graph* that colors constant-degree graph with $(\Delta+1)$ colors is presented in Figure 4.4. The algorithm consists of two phases. In the first phase we iteratively call the *Find-Forest* procedure, each time removing the edges of the constructed forest. This phase continues until no edges remain, at which point we color all the vertices with one color.

In the second phase we iteratively add the edges of the forests back into the graph, each time recoloring the vertices to maintain a consistent coloring. At the beginning of each iteration of this phase, the edges of the current forest ($E'$) are

added, making the existing $(\Delta+1)$-coloring inconsistent. This forest is colored with 3 colors using the *3-Color-Rooted-Tree* procedure. Now, each vertex has two colors – one from the coloring at the previous iteration and one from the coloring of the forest. The pairs of colors form a valid $3(\Delta+1)$-coloring of the graph. The iteration finishes by enumerating the color classes, recoloring each vertex of the current color with a color from $\{0,\ldots,\Delta\}$ that is different from the colors of its neighbors (note that we can recolor all the vertices of the same color in parallel because they are independent).

In the proof of the following theorem, we assume that $\Delta = O(\log n)$. Under this assumption, we can use word operations to implement, in constant time, union and membership testing operations on sets of size $\Delta + 1$.

**Theorem 4.4.1** *The algorithm Color-Constant-Degree-Graph runs in* $O(\Delta \log \Delta(\Delta + \lg^* n))$ *time and colors the graph with* $(\Delta+1)$ *colors.*

*Proof*: At each iteration all edges of a spanning forest are removed. From the above discussion it follows that each vertex that still has neighbors in the beginning of an iteration has at least one edge removed during that iteration and therefore its degree decreases. Hence, the first phase of the algorithm terminates in at most $\Delta$ iterations.

The second phase terminates in at most $\Delta$ iterations as well. Each iteration consists of two stages. First, the current forest is colored using procedure *3-Color-Rooted-Tree*, which takes, by theorem 4.3.2, $O(\log \Delta \lg^* n)$ time on an EREW PRAM (the $\log \Delta$ factor appears because we do not use the concurrent-read capability). Now we iterate over all the colors. Each iteration can be done in $O(\log \Delta)$ time using word operations. Hence, one iteration of the second phase takes $O(\log \Delta \lg^* n + \Delta \log \Delta)$ time, leading to an overall $O(\Delta \log \Delta(\Delta + \lg^* n))$ running time on an EREW PRAM. ∎

Having a $(\Delta+1)$-coloring of a graph enables us to find an MIS in this graph. The following theorem states this fact formally. (We refer to the algorithm described in the proof as *Constant-Degree-MIS* in the subsequent sections.)

**Theorem 4.4.2** *An MIS in constant-degree graphs can be found in $O(\lg^* n)$ time on an EREW PRAM using $O(n)$ processors.*

*Proof*: After coloring the graph in a constant number of colors using the procedure *Color-Constant-Degree-Graph*, one can find an MIS by iterating over the colors, taking all the remaining vertices of the current color, adding them to the independent set, and removing them and all their neighbors from the graph. By theorem 4.4.1, the coloring of a constant-degree graph takes $O(\lg^* n)$ time on an EREW PRAM. The selection of all vertices with a specific color and the removal of all neighbors of the selected vertices takes constant time. ∎

The proofs of theorems 4.4.1 and 4.4.2 also imply that the algorithms *Color-Constant-Degree-Graph* and *Constant-Degree-MIS* have a polylogarithmic running times for graphs with polylogarithmic maximum degrees. In this case, however, the assumption that the word size is greater than $\Delta$ is unreasonable, so the running time of the algorithms becomes $O(\Delta(\Delta^2 + \log \Delta \lg^* n))$.

The above algorithms can be implemented in the distributed model of computation [31,2], where processors have fixed connections determined by the input graph. The algorithms in the distributed model achieve the same $O(\lg^* n)$ bound as in the EREW PRAM model. Linial has recently shown [54] that $\Omega(\lg^* n)$ time is required in the distributed model to find a maximal independent set on a chain. Our algorithms are therefore optimal (to within a constant factor) in the distributed model.

## 4.5 Algorithms for Planar Graphs

In this section we study the problem of coloring a planar graph and the problem of finding a maximal matching in a planar graph.

Any planar graph can be 4-colored. Linear-time sequential algorithms, however, are known only for 5-coloring planar graphs. In this section we describe a simple and

```
Procedure 7-Color-Planar-Graph.
    V' ← V;
    V_1, V_2, ... V_{lg n} ← ∅;
    i ← 0;
    ⟨⟨ first stage⟩⟩
    while V' ≠ ∅ for all v ∈ V' in parallel do begin
          if Degree(v) ≤ 6 then begin
             V_i ← V_i + {v};
             V' ← V' − {v};
          end;
          i ← i + 1;
    end;
    ⟨⟨ second stage ⟩⟩
    for  i ← i − 1 to 0 do
         while V_i ≠ ∅ do begin
               E_i ← {(v, w) | v, w ∈ V_i ; (v, w) ∈ E};
               I ← Constant-Degree-MIS(V_i, E_i);
               for all v ∈ I in parallel do
                    C_v ← max_{i∈{{1...7}−{C_w | w∈V';(v,w)∈E}}}(i);
               V' ← V' + I;
               V_i ← V_i − I;
         end;
end.
```

Figure 4.5: The algorithm for 7-coloring of planar graphs

efficient parallel algorithm that 7-colors a planar graph, and show how to construct a more complicated parallel algorithm to 5-color a planar graph.

First we describe an algorithm for 7-coloring of planar graphs. The algorithm, called *7-Color-Planar-Graph*, is shown in Figure 4.5. The algorithm consists of two stages. In the first stage, we iteratively partition the vertices of the graph into layers. At each iteration we create a new layer consisting of all vertices of the graph with degree 6 or less and delete these vertices from the graph.

The second stage returns the layers to the graph in the order opposite to the order in which the layers are removed. After a layer is returned, it is 7-colored in a way consistent with the coloring of the layers which have been returned and colored

in the previous iterations. Note that all the vertices of the returned layer have a degree of at most 6 in the current graph.

The layer is colored by iteratively applying the *Constant-Degree-MIS* procedure to find an MIS in the subgraph induced by the uncolored vertices of the layer, and coloring each of the selected vertices in a color different from its colored neighbors. Since the uncolored vertices have a degree of at most 6 in the current graph, we never need more than 7 colors.

**Theorem 4.5.1** *The algorithm 7-Color-Planar-Graph runs in* $O(\log n \lg^* n)$ *time on a CRCW PRAM and in* $O(\log^2 n)$ *time on an EREW PRAM.*

*Proof*: In a planar graph, at least a constant fraction (1/7) of the vertices have a degree less or equal to 6, and therefore the first stage of the *7-Color-Planar-Graph* algorithm terminates in at most $O(\log n)$ steps. At each step we have to identify the vertices that have degree less than 7 in the remaining graph. This takes constant time on a CRCW PRAM (assuming that if two or more processors simultaneously write into some location, one of them will succeed) and $O(\log n)$ time on an EREW PRAM.

In the second stage all the uncolored vertices are of degree less or equal to 6 and therefore, by theorem 4.4.2, the procedure *Constant-Degree-MIS* finds, in $O(\lg^* n)$ time, an MIS in the graph induced by these vertices. When the algorithm colors a maximal independent set, at least one uncolored neighbor of each uncolored vertex becomes colored. Therefore the second part of the second stage terminates in at most 7 iterations.

Since the first stage takes $O(\log n)$ time on a CRCW PRAM and $O(\log^2 n)$ time on an EREW PRAM, and since each one of the $O(\log n)$ iterations of the second stage is dominated by a call to *Constant-Degree-MIS*, the total running time is $O(\log n \lg^* n)$ on a CRCW PRAM and $O(\log^2 n)$ on an EREW PRAM. ∎

*Remark*: If, at each stage, instead of removing from the graph all the vertices with degree less than 6, we remove all the vertices with degree less or equal to the average

degree, the algorithm described above produces a correct result in polylogarithmic time for any graph $G$ such that the average degree of any vertex-induced subgraph $G'$ of $G$ is polylogarithmic in the size of $G'$. This class contains many important subclasses including graphs that are unions of a polylogarithmic number of planar graphs (i.e. graphs with polylogarithmic thickness).

Our techniques together with the ideas presented in [10] can be used to construct a deterministic $O(\log^3 n \lg^* n)$ time algorithm for 5-coloring a planar graph. The 5-coloring algorithm has two stages. The first stage of the algorithm partitions the graph into layers such that vertices in any layer are independent and have degree of at most 6 in the graph induced by the vertices in its layer and the higher numbered layers. This partitioning can be done by partitioning the graph into layers as in the first stage of the 7-coloring algorithm, coloring each layer in 7 colors, and taking each color class to be the new layer. The second stage of the algorithm adds layers one by one, starting from the layer with the highest number, each time recoloring the graph with 5 colors.

Before describing the second stage, we need the following definitions. Let $G$ be a partially colored graph and let $c_1$ and $c_2$ be two distinct colors. A *color component* is a connected component of a subgraph of $G$ induced by all vertices of color $c_1$ and $c_2$. A *color component flip* is a recoloring of the color component that exchanges colors $c_1$ and $c_2$. A color component flip does not affect the validity of coloring.

We can proceed with the description of the second stage of the algorithm. After a layer is added to already colored graph, we first color all vertices that can be colored without changing the existing coloring. This can be done in the same way as in the 7-coloring algorithm. Now all 5 colors are represented among neighbors of each uncolored vertex. Since the uncolored vertices have degree of at most 6, the results of [10] imply that for every uncolored vertex $v$ there are two colors $c_1$ and $c_2$ such that $v$ has exactly one neighbor $w_1$ of color $c_1$ and exactly one neighbor $w_2$ of color $c_2$. Furthermore, the vertices $w_1$ and $w_2$ belong to different color components induced by colors $c_1$ and $c_2$. Flipping each one of these color component allows us to color $v$. The problem is, however, that flipping both color components simultaneously does not allow us to color $v$. We call such color components *dependent*.

Where as Boyar and Karloff use randomness to deal with these dependencies, we use our symmetry-breaking techniques as follows. For each pair of distinct colors $c_1$ and $c_2$, we construct color components induced by these colors. Then we construct a *dependency graph* with vertices corresponding to the color components and edges corresponding to the dependencies between the color components. Flipping a set of color components that corresponds to an independent set in the dependency graph does not cause conflicts. Suppose we can find an independent set in the dependency graph such that flipping the corresponding set of color components allows us to color a constant fraction of uncolored vertices. Then in $O(\log n)$ iterations will be able to color all uncolored vertices.

We find such an independent set in the dependency graph as follows. Observe that the dependency graph is planar, so we can 7-color this graph using the *7-Color-Planar-Graph* algorithm. Then, for each pair of distinct colors and for each color class of the corresponding dependency graph, we compute the number of uncolored vertices of the original graph which can be colored if the color components corresponding to vertices in the color class are flipped. For each of the 10 possible choices of colors $c_1$ and $c_2$ there are 7 color classes, so the total number of times that we count the number of vertices that can be colored if a color class is flipped is 70. Since each uncolored vertex is counted at least once, there is a color class such that flipping all color components in this class allows us to color at least 1/70 of all uncolored vertices.

Now we analyze to complexity of the algorithm. The outer loop of the algorithm that iterates over layers is executed $O(\log n)$ times, and the inner loop that colors a constant fraction of uncolored vertices is executed $O(\log n)$ times as well. Each iteration of the inner loop does 10 connected component computations, 70 enumerations, and 10 calls to the *7-Color-Planar-Graph* procedure. Since a connected component computation can be done in $O(\log n)$ time on CRCW PRAM using Shiloach-Vishkin algorithm [65] and an enumeration can be done in $O(\log n)$ time by using parallel prefix computations (see Section 3.2), the *7-Color-Planar-Graph* procedure, which runs in $O(\log n \lg^* n)$ time is the bottleneck of the inner loop. The overall running time of the algorithm is $O(\log^3 n \lg^* n)$.

The above result is summarized in the following theorem.

**Theorem 4.5.2** *A planar graph can be 5-colored in $O(\log^3 n)$ time on a CRCW PRAM using $O(n)$ processors.*

Using the techniques described in this chapter it is easy to construct a fast algorithm for finding a maximal matching in a planar graph.

**Theorem 4.5.3** *A maximal matching in planar graph can be found in $O(\log n \lg^* n)$ time on a CRCW PRAM.*

*Proof*: First, the algorithm partitions the graph into layers, such that the vertices in a layer are of degree at most 6 in the graph induced by the vertices of this layer and higher-numbered layers. The algorithm proceeds by iteratively adding a layer, finding a maximal matching in the obtained graph, and removing the end-points of the edges in the matching. At the end of each iteration the remaining vertices induce a graph of degree zero and therefore at the beginning of each iteration the maximum degree of the induced graph is 6. Hence, a maximal matching in this graph can be found in $O(\lg^* n)$ time by finding a maximal independent set in the line-graph, which also has a constant maximum degree. Each iteration takes $O(\lg^* n)$ time on a CRCW PRAM and the number of iterations is $O(\log n)$. This gives $O(\log n \lg^* n)$ total running time. ∎

## 4.6 Lower Bounds

In this section we prove two lower bounds for a CRCW PRAM with polynomial number of processors:

- Finding a MIS in a general graph takes $\Omega(\log n / \operatorname{loglog} n)$ time.
- 2-coloring a directed list takes $\Omega(\log n / \operatorname{loglog} n)$ time.

The first lower bound complements the $O(\log n)$ CRCW PRAM upper bound for the MIS problem that is achieved by Luby's algorithm [55]. The second lower bound complements Theorem 4.3.2 in this chapter.

We need to define the following two functions. The PARITY function on $n$ bits $x_1 \ldots x_n$ equals $(x_1 + \ldots + x_n) \bmod 2$. The MAJORITY function on $n$ bits equals 1 if $x_1 + \ldots + x_n > n/2$, and equals to 0 otherwise. The results of [25,4,5] imply that any CRCW PRAM algorithm for PARITY or MAJORITY that uses a polynomial number of processors requires $\Omega(\log n / \log\log n)$ time.

**Theorem 4.6.1** *The running time of any MIS algorithm on a CRCW PRAM with a polynomial number of processors is* $\Omega(\log n / \log\log n)$.

*Proof*: Given an instance of MAJORITY, we construct an instance of MIS in constant CRCW PRAM time. Let $x_1, x_2, \ldots, x_n$ be an instance of MAJORITY. We construct a complete bipartite graph $G = (V, E)$ with vertices corresponding to zero bits of the input on one side and vertices corresponding to one bits on the other side. The graph $G = (V, E)$ is defined by

$$\begin{aligned} V &= \{1, \ldots, n\} \\ E &= \{(i,j) \mid x_i \neq x_j\}. \end{aligned}$$

To construct this graph, assign a processor $P_{ij}$ for each pair $1 \leq i < j \leq n$. Then, each processor $P_{ij}$ writes 1 into location $M_{ij}$ if $x_i \neq x_j$ and 0 otherwise.

A maximal matching in a complete bipartite graph is also a maximum one. By constructing a maximal independent set in the line-graph $G'$ of $G$, one can find a maximal matching in $G$. To construct the graph $G'$ assign a processor $P_{ijk}$ for each distinct $i, j, k \leq n$. Each $P_{ijk}$ writes 1 into location $M_{(i,j),(j,k)}$ if $M_{ij} = M_{jk} = 1$ and 0 otherwise.

The MAJORITY function equals 1 if and only if there is an unmatched vertex $i \in G$ such that $x_i = 1$, which can be checked on a CRCW PRAM in constant time. ∎

**Theorem 4.6.2** *The time to 2-color a directed list on a CRCW PRAM with a polynomial number of processors is* $\Omega(\log n / \log\log n)$.

*Proof*: We show a constant-time reduction from PARITY to the 2-coloring of a directed list. First, we show how to construct, in constant time, a directed list with elements corresponding to all the input bits $x_i$ with value of 1. Let $x_1, x_2, \ldots, x_n$ be an instance of PARITY. Associate a processor $P_i$ with each input cell $M_i$ that initially holds the value of $x_i$. Associate a set of processors $P_i^{jk}$ with each index $i$, $1 \leq k \leq j < i$. In one step, each processor $P_i^{jk}$ reads the value of $M_k$ and, if it equals 1, writes 1 into $M_i^j$, effectively computing the OR-function on the input values $x_{i-j}, x_{i-j+1}, \ldots, x_{i-1}$. Assign a processor $P_i^j$ to each $M_i^j$. Each processor $P_i^j$ reads $M_i^j$ and $M_i^{j+1}$ and writes $j$ into $M_i'$ if and only if $M_i^j \neq M_i^{j+1}$. It can be seen that for all $0 \leq i \leq n$, $M_i'$ holds $\max\{j \mid j < i, x_j = 1\}$.

We have constructed a directed list with elements corresponding to all the input bits $x_i$ with value of 1. Assume this list is 2-colored. Then PARITY equals 1 if and only if both ends of the list are colored in the same color, which can be checked in constant time. ∎

## 4.7 Remarks

In this chapter we described the technique for 3-coloring rooted trees and showed how this technique can be used to design parallel algorithms. Our technique applies when we have a set of jobs each of which can be performed at this time, but simultaneous execution of these jobs may cause conflicts. As we have seen, if the graph describing these conflicts is sparse, the technique allows to select a large subset of non-conflicting jobs. Note that there is no need for such a technique in the sequential framework. This chapter shows how important it is to identify the fundamental problems of parallel computation and to developed efficient solutions to these problems.

# Conclusion

In the first chapter of this thesis, we introduced a new approach to the maximum flow problem. This approach combines the concept of preflow due to Karzanov with a novel concept of distance labeling. Using our approach, we have developed better algorithms for the problem, including an $O(nm \log(n^2/m))$ time sequential algorithm and $O(n^2 \log n)$ time parallel algorithm. The parallel algorithm uses a linear amount of memory and, as we have seen in Chapter 3, is practical.

In the second chapter, we developed a framework for solving minimum-cost flow problems. This framework permits the extentions of maximum flow techniques to the more general minimum-cost flow problem. We generalized the techniques developed in Chapter 1 to obtain an $O(nm \log(n) \log(nC))$ time sequential algorithm. We also believe that a generalization of these techniques leads to an $O(nm \log(n^2/m) \log(nC))$ time sequential algorithm and an $O(n^2 \log(n) \log(nC))$ time parallel algorithm that uses a linear amount of memory, but the bounds we prove in Chapter 2 are somewhat weaker. We also show how to incorporate in our framework Dinic's blocking flow method for solving maximum flow problems. Using the algorithms for finding blocking flows in acyclic networks, we obtain sequential algorithms for the minimum-cost flow problem that run in $O(nm \log(n) \log(nC))$, $O(n^{5/3} m^{2/3} \log(nC))$, and $O(n^3 \log(nC))$ time, and an $O(n^2 \log(n) \log(nC))$ time parallel algorithm that uses $O(n^2)$ memory.

In the third chapter we described an implementation of the parallel maximum flow algorithm. The key feature of this implementation is the use of parallel prefix operations as primitives, which enables us to obtain an efficient implementation while maintaining a high level of abstraction. The usefulness of the parallel prefix operations stimulated our search for other fundamental techniques for the design and implementation of parallel algorithms. In Chapter 4 we introduced parallel symmetry-breaking techniques and showed how these techniques can be applied to

design efficient algorithms. We also showed lower bounds which imply that our symmetry-breaking techniques are optimal in some contexts.

In conclusion, we would like to discuss potential improvements and extensions of the results of this thesis. As discussed in Section 1.7, an interesting question related to the maximum flow problem is the existence of an $O(nm)$ algorithm for the problem. For the special case of zero-one networks, Even and Tarjan [17] show that Dinic's algorithm is more efficient than for the general networks. Can their bounds be improved using the approach of Chapter 1? The improved bounds would imply a better bound for the maximum bipartite matching problem. For the minimum-cost flow problem, a promising approach is scaling by value (i.e., cost-capacity product), as discussed in Section 2.11.

The maximum flow problem and the minimum-cost flow problems have been studied extensively. The multicommodity flow problem, on the other hand, has received relatively little attention. The current algorithms for this problems [43,45] use the techniques of mathematical programming. An efficient multicommodity flow algorithm that uses graph-theoretic techniques similar to the techniques discussed in this thesis would be a very interesting result.

In the area of parallel algorithms, one of the most important problems is that of identifying the fundamental problems related to parallel computation and finding efficient solutions for these problems.

This thesis makes a step towards a better understanding of algorithms on graphs, both in sequential and parallel contexts. It is our hope that the techniques and approaches presented here will prove useful in further advances in this important field.

# Bibliography


[1] R. K. Ahuja and J. B. Orlin. A simple $O(nm+n^2 \log c_{max})$ sequential algorithm for the maximum flow problem. 1986. Unpublished manuscript.

[2] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32:804–823, 1985.

[3] C. A. Bateson. Performance comparison of two algorithms for weighted bipartite matchings. 1985. M.S. thesis, University of Colorado, Boulder, Colorado.

[4] P. Beame. *Lower Bounds in Parallel Machine Computation.* PhD thesis, University of Toronto, Toronto, Canada, 1986.

[5] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. 19th ACM Symp. on Theory of Computing*, 1987. (To appear).

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

[7] D. P. Bertsekas and P. Tseng. Relaxation methods for minimum cost ordinary and generalized network flow problems. *O. R. Journal*, 1986. (To appear).

[8] R. G. Bland and D. L. Jensen. *On the Computational Behavior of a Polynomial-Time Network Flow Algorithm.* Technical Report 661, School of Operations Research and Industrial Engineering, Cornell University, 1985.

[9] G. Blelloch. *Parallel Prefix vs. Concurrent Memory Access.* Technical Report, Thinking Machines, Inc., 1986.

[10] J. Boyar and H. J. Karloff. Coloring planar graphs in parallel. *J. of Algorithms*, 1979. (To appear).

[11] A. K. Chandra, L. J. Stockmeyer, and U. Vishkin. A complexity theory for unbounded fan-in parallelism. In *Proc. 23rd IEEE Symp. on Foundations of Comp. Science*, pages 1–13, 1982.

[12] R. V. Cherkasky. Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations. *Mathematical Methods of Solution of Economical Problems*, 7:112–125, 1977. (In Russian).

[13] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 206–219, 1986.

[14] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dok.*, 11:1277–1280, 1970.

[15] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[16] S. Even. *Graph Algorithms.* Computer Science Press, Potomac, MD, 1979.

[17] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. of Computing*, 4:507–518, 1975.

[18] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 9:948–960, 1972.

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

[20] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Math.*, 8:399–404, 1956.

[21] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[22] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses. In *Proc. 25th IEEE Symp. on Foundations of Computer Science*, pages 338–346, 1984.

[23] S. Fujishige. *An $O(m^3 \log n)$ Capacity-Rounding Algorithm for the Minimum-Cost Circulation Problem: a Dual Framework of the Tardos Algorithm.* Technical Report 253, University of Tsukuba, Japan, 1985.

[24] D. R. Fulkerson. An out-of-kilter method for minimal cost flow problems. *SIAM J. Appl. Math*, 9:18–27, 1961.

[25] M. Furst, J. Saxe, and M. Sipser. Parity,circuits, and the polynomial time hierarchy. In *Proc. 22nd IEEE Conf on Foundations of Computer Science*, pages 260–270, 1981.

[26] H. N. Gabow. Scaling algorithms for network problems. *J. of Comp. and Sys. Sci.*, 31:148–168, 1985.

[27] Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Informatica*, 14:221–242, 1980.

[28] Z. Galil. Personal communication. 1987.

[29] Z. Galil and A. Naamad. An $O(EV^{5/3} \log^2 V)$ algorithm for the maximal flow problem. *J. Comput. System Sci.*, 21:203–217, 1980.

[30] Z. Galil and E. Tardos. An $O(n^2 \log n(m + n \log n))$ min-cost flow algorithm. In *Proc. 27th IEEE Symp. of Foundations of Computer Science*, pages 1–9, 1986.

[31] R. G. Gallager, P. A. Humblet, and P. M Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[32] S. I. Gass. *Linear Programming: Methods and Applications.* McGraw-Hill, 1958.

[33] A. V. Goldberg. *A New Max-Flow Algorithm.* Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.

[34] A. V. Goldberg and S. A. Plotkin. *Efficient Parallel Algorithms for* $(\Delta + 1)$*-Coloring and Maximal Independent Set Problems.* Technical Report MIT/LCS/TM-320, MIT, 1987.

[35] A. V. Goldberg and S. A. Plotkin. Parallel $(\Delta + 1)$ coloring of constant-degree graphs. *Information Process. Lett.*, 1987. (To appear).

[36] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. 19th ACM Symp. on Theory of Computing*, 1987. (To appear).

[37] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 136–146, 1986.

[38] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proc. 19th ACM Symp. on Theory of Computing*, 1987. (To appear).

[39] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independent set problem. 1986. Unpublished manuscript.

[40] A. Gottlieb, R. Grishman, C. P. Kruskal, K. M. McAuliffe, L. Rudolf, and M. Snir. The NYU ultracomputer – designing a MIMD shared memory parallel computer. *IEEE Trans. Comput*, 32:175–189, 1983.

[41] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

[42] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Comm. of the ACM*, 29:1170–1183, 1986.

[43] T. C. Hu. *Integer Programming and Network Flows.* Addison-Wesley, Reading, MA, 1969.

[44] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Information Proc. Lett.*, 22:57–60, 1986.

[45] S. Kapoor and P. M. Vaidya. Fast algorithms for convex quadratic programming and multicommodity flows. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 147–159, 1986.

[46] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 266–272, 1984.

[47] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dok.*, 15:434–437, 1974.

[48] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[49] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. In *Proc. Int. Conf. on Parallel Processing*, pages 180–185, 1985.

[50] H. W. Kuhn. Variants of the Hungarian method for assignment problem. *Naval Res. Loyist. Quart.*, 3:253–258, 1956.

[51] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. Assoc. Comp. Mach.*, 27:831–838, 1980.

[52] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart, and Winston, New York, NY., 1976.

[53] C. Leiserson and B. Maggs. Communication-efficient parallel graph algorithms. In *Proc. of International Conference on Parallel Processing*, pages 861–868, 1986.

[54] N. Linial. Personal communication. 1986.

[55] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 1–10, 1985.

[56] V. M. Malhotra, M. Pramodh Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inform. Process. Lett.*, 7:277–278, 1978.

[57] G. J. Minty. Monotone networks. *Proc. Roy. Soc. London*, A(257):194–212, 1960.

[58] A. T. Ogielski. Integer optimization and zero-temperature fixed point in Ising random-field systems. *Physical Review Lett.*, 57(10):1251–1254, 1986.

[59] J. B. Orlin. *Genuinely Polynomial Simplex and Non-Simplex Algorithms for the Minimum Cost Flow Problem.* Technical Report No. 1615-84, Sloan School of Management, MIT, December 1984.

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

[61] J. C. Picard and H. D. Ratliff. Minimum cuts and related problems. *Networks*, 5:357–370, 1975.

[62] H. Röck. Scaling techniques for minimal cost network flows. In V. Page, editor, *Discrete Structures and Algorithms*, Carl Hansen, Munich, 1980.

[63] G. Shannon. Reduction techniques for designing linear-processor parallel algorithms on sparse graphs. 1986. Unpublished manuscript.

[64] Y. Shiloach. *An $O(nI\log^2 I)$ Maximum-Flow Algorithm.* Technical Report STAN-CS-78-802, Computer Science Department, Stanford University, Stanford, CA, 1978.

[65] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[66] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms*, 3:128–146, 1982.

[67] D. D. Sleator. *An $O(nm \log n)$ Algorithm for Maximum Network Flow.* Technical Report STAN-CS-80-831, Computer Science Department, Stanford University, Stanford, CA, 1980.

[68] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[69] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[70] E. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

[71] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[72] R. E. Tarjan. A simple version of Karzanov's blocking flow algorithm. *Operations Research Letters*, 2:265–268, 1984.

[73] L. G. Valiant. Parallel computation. In *Proc. 7th IBM Symp. on Mathemetical Foundations of Computer Science*, 1982.