SCHOOL OF OPERATIONS RESEARCH
AND INDUSTRIAL ENGINEERING
COLLEGE OF ENGINEERING
CORNELL UNIVERSITY
ITHACA, NEW YORK 14853-7501

TECHNICAL REPORT NO. 860

September 1989

NETWORK FLOW ALGORITHMS

Andrew V. Goldberg[1]
Éva Tardos[2]
Robert E. Tarjan[3]

[1]Department of Computer Science, Stanford University, Stanford, CA 94305.
Research partially supported by NSF Presidential Young Investigator Grant CCR-8858097, IBM Faculty Development Award, ONR Contract N00014-88-K-0166, and AT&T Bell Laboratories.

[2]School of OR&IE, Cornell University, Ithaca, NY 14853.
Research partially supported by Air Force contract AFOSR-86-0078.

[3]Department of Computer Science, Princeton University, Princeton, NJ 08544 and AT&T Bell Laboratories, Murray Hill, NJ 07974.
Research partially supported by the National Science Foundation, Grant No. DCR-8605961, and the Office of Naval Research, Contract No. N00014-87-K-0467.

Contents

Introduction **3**

1 Preliminaries **9**

1.1 Flows and Residual Graphs 9

1.2 The Maximum Flow Problem 10

1.3 The Minimum-Cost Circulation Problem 10

1.4 The Transshipment Problem 12

1.5 The Generalized Flow Problem 14

1.6 The Restricted Problem 16

1.7 Decomposition Theorems 17

2 The Maximum Flow Problem **19**

2.1 Introduction 19

2.2 A Generic Algorithm 22

2.3 Efficient Implementation 25

2.4 Excess Scaling 28

2.5 Use of Dynamic Trees 30

3 The Minimum-Cost Circulation Problem: Cost-Scaling **33**

3.1 Introduction 33

3.2 Approximate Optimality 34

3.3 The Generalized Cost-Scaling Framework 35

3.4 A Generic Refinement Algorithm 35

3.5 Efficient Implementation 39

3.6 Refinement Using Blocking Flows 41

4 Strongly Polynomial Algorithms Based on Cost-Scaling **45**

4.1 Introduction 45

4.2 Fitting Price Functions and Tight Error Parameters 45

4.3 Fixed Arcs 47

4.4 The Strongly Polynomial Framework 49

4.5 Cycle-Canceling Algorithms 50

5 Capacity-Scaling Algorithms **55**

5.1 Introduction 55

5.2 The Edmonds-Karp Algorithm 56

5.3 Strongly Polynomial Algorithms 58

6 The Generalized Flow Problem **63**

6.1 Introduction 63

6.2 Simple Combinatorial Algorithms 64

6.3 Polynomial-Time Combinatorial Algorithms 66

Bibliography **72**

Introduction

Network flow problems are central problems in operations research, computer science, and engineering and they arise in many real world applications. Starting with early work in linear programming and spurred by the classic book of Ford and Fulkerson [25], the study of such problems has led to continuing improvements in the efficiency of network flow algorithms. In spite of the long history of this study, many substantial results have been obtained within the last several years. In this survey we examine some of these recent developments and the ideas behind them.

We discuss the classical network flow problems, the *maximum flow problem* and the *minimum-cost circulation problem*, and a less standard problem, the *generalized flow problem*, sometimes called the problem of *flows with losses and gains*. The survey contains six chapters in addition to this introduction. Chapter 1 develops the terminology needed to discuss network flow problems. Chapter 2 discusses the maximum flow problem. Chapters 3, 4, and 5 discuss different aspects of the minimum-cost circulation problem, and Chapter 6 discusses the generalized flow problem. In the remainder of this introduction, we mention some of the history of network flow research, comment on some of the results to be presented in detail in later sections, and mention some results not covered in this survey.

We are interested in algorithms whose running time is small as a function of the size of the network and the numbers involved (e.g. capacities, costs, or gains). As a measure of the network size, we use n to denote the number of vertices and m to denote the number of arcs. As measures of the number sizes, we use U to denote the maximum arc capacity, C to denote the maximum arc cost, and B (in the case of the generalized flow problem) to denote the maximum numerator or denominator of the arc capacities and gains. In bounds using U, C, or B, we make the assumption that the capacities and costs are integers, and that the gains in the generalized flow problem are rational numbers.

We are most interested in polynomial-time algorithms. We make the following distinctions. An algorithm is *pseudopolynomial* if its running time has a bound that is polynomial in n, m, and the appropriate subset of U, C, and B. An algorithm is *polynomial* if its running time has a bound that is polynomial in n, m, and the appropriate subset of $\log U, \log C$, and $\log B$[1]. An

[1] All logarithms in this paper without an explicit base are base two.

algorithm is *strongly polynomial* if its running time has a bound that is polynomial in n and m[2]. When comparing polynomial algorithms to strongly polynomial ones we shall use the *similarity assumption* that $\log U$, $\log C$ and $\log B$ are $\Theta(\log n)$ [31]. We shall also be interested in strongly polynomial algorithms, however.

The network flow problems discussed in this survey are special cases of linear programming, and thus they can be solved by general-purpose linear programming algorithms. However, the combinatorial structure of these problems makes it possible to obtain more efficient algorithms.

We shall not discuss in detail algorithms that are based on general linear programming methods. We should mention, however, that the first algorithm designed for network flow problems was the *network simplex method* of Dantzig [19]. It is a variant of the linear programming simplex method designed to take advantage of the combinatorial structure of network flow problems. Variants of the simplex method that avoid cycling give an exponential bound on the complexity of all the network flow problems. (Cunningham [18] gives an elegant anti-cycling strategy for the network simplex method based on graph-theoretic properties of the minimum-cost circulation problem). Recently, Goldfarb and Hao [50] have designed a variant of the primal network simplex method for the maximum flow problem that runs in strongly polynomial time (see Section 2.1). Orlin [80] designed a variant of the dual network simplex method for the minimum-cost circulation problem that runs in strongly polynomial time (see Chapter 5). For a long time, the network simplex method has been the method of choice in practice, in particular for the minimum-cost circulation problem (see e.g. [52]); for large instances of hard problems, the new scaling algorithms are probably better, however.

The first pseudopolynomial algorithm for the maximum flow problem is the *augmenting path* algorithm of Ford and Fulkerson [26, 25]. Dinic [20] and Edmonds and Karp [21] independently obtained polynomial versions of the augmenting path algorithm. Since then, several more-efficient algorithms have been developed. Chapter 2 presents the *push/relabel method*, recently proposed by Goldberg [38] and Goldberg and Tarjan [43], along with some of its more efficient variants.

The first pseudopolynomial algorithm for the minimum-cost circulation problem is the *out-of-kilter* method, which was developed independently by Yakovleva [103], Minty [75], and Fulkerson [30]. The first polynomial algorithm for the minimum-cost circulation problem is due to Edmonds and Karp [21]. To develop this algorithm Edmonds and Karp introduced the technique of *scaling*, which has proved to be a useful tool in the design and analysis of algorithms for a variety of combinatorial optimization problems. Chapter 3 and Section 5.2 are devoted to scaling algorithms for the minimum-cost circulation problem.

The maximum flow algorithms of Dinic [20] and Edmonds and Karp [21] are strongly polynomial, but the minimum-cost circulation algorithm of Edmonds and Karp [21] is not. The first strongly polynomial algorithm for the minimum-cost circulation problem was designed by Tar-

[2]For a more formal definition of polynomial and strongly polynomial algorithms, see [53].

dos [94]. Chapter 4 and Section 5.3 are devoted to recent strongly polynomial algorithms for the minimum-cost circulation problem.

The first augmenting path algorithms for the generalized flow problem were developed independently by Jewell [60, 61] and Onaga [77]. Many pseudopolynomial minimum-cost circulation algorithms have been adapted for the generalized flow problem (see [100] for a survey). The first polynomial-time algorithm for the generalized flow problem was the ellipsoid method [68]. Kapoor and Vaidya [63] have shown how to speed up Karmarkar [64] — or Renegar [87] — type interior-point algorithms on network flow problems by taking advantage of the special structure of the matrices used in the linear programming formulations of these problems. Vaidya's algorithm [101] is the fastest currently known algorithm for the generalized flow problem. The first polynomial algorithms for the generalized flow problem that are not based on general-purpose linear programming methods are due to Goldberg, Plotkin, and Tardos [41]. These algorithms are discussed in Chapter 6. The existence of a strongly polynomial algorithm for the generalized flow problem is an interesting open question.

Important special cases of network flow problems that will not be covered in this survey are the *bipartite matching* problem and its weighted version, the *assignment* problem. These problems can be stated as maximum flow and minimum-cost circulation problems, respectively, on networks with unit capacities and a special structure (see e.g. [23, 97]). Some of the efficient algorithms for the more general problems have evolved from efficient algorithms developed earlier for these simpler problems.

König's [70] proof of a good characterization of the maximum size of a matching in a bipartite graph gives an $O(nm)$–time algorithm for finding a maximum matching. The Ford-Fulkerson maximum flow algorithm can be viewed as an extension of this algorithm. Hopcroft and Karp [56] gave an $O(\sqrt{n}m)$ algorithm for the bipartite matching problem. Even and Tarjan observed [24] that Dinic's maximum flow algorithm, when applied to the bipartite matching problem, behaves similarly to the Hopcroft-Karp algorithm and runs in $O(\sqrt{n}m)$ time as well. A variation of the Goldberg-Tarjan maximum flow algorithm (which can be viewed as a generalization of Dinic's algorithm) can be easily shown to lead to the same bound [5, 82]. In spite of recent progress on related problems, the $O(\sqrt{n}m)$ bound has not been improved.

The first algorithm for the assignment problem is the *Hungarian method* of Kuhn [71]. The out-of-kilter algorithm is an extension of this algorithm to the minimum-cost circulation problem. The Hungarian method solves the assignment problem in $O(n)$ shortest path computations. Edmonds and Karp [21] and Tomizawa [99] have observed that the dual variables can be maintained so that these shortest path computations are on graphs with non-negative arc costs. Combined with the shortest path algorithm of [28], this observation gives an $O(n(m + n \log n))$ bound for the problem. Gabow [31] used scaling to obtain an $O(n^{3/4}m \log C)$ algorithm for the problem. Extending ideas of the Hopcroft-Karp bipartite matching algorithm and those of the Goldberg-Tarjan minimum-cost

Problem	Date	Discoverer	Running Time	References
Bipartite Matching	1973	Hopcroft and Karp	$O(\sqrt{n}m)$	[56]
Assignment	1955 1987	Kuhn Gabow and Tarjan	$O(n(m + n\log n))$ $O(\sqrt{n}m\log(nC))$	[71] [33]
Maximum Flow	1986 1988	Goldberg and Tarjan Ahuja, Orlin, and Tarjan	$O(nm\log(n^2/m))$ $O(nm\log(\frac{n}{m}\sqrt{\log U} + 2))$	[43] [4]
Minimum-Cost Circulation	1972 1987 1987 1988	Edmonds and Karp Goldberg and Tarjan Orlin Ahuja, Goldberg, Orlin, and Tarjan	$O(m\log U(m + n\log n))$ $O(nm\log(n^2/m)\log(nC))$ $O(m\log n(m + n\log n))$ $O(nm\log\log U\log(nC))$	[21] [47] [79] [1]
Generalized Flow	1989	Vaidya	$O(n^2m^{1.5}\log(nB))$	[101]

Table 1: **Fastest currently known algorithms for network flow problems.**

circulation algorithm (discussed in Section 3), Gabow and Tarjan obtained an $O(\sqrt{n}m\log(nC))$ algorithm for the assignment problem.

A more recent pseudopolynomial algorithm for the assignment problem is the auction algorithm of Bertsekas [9] (first published in [10]). This algorithm contains some of the elements of the push/relabel algorithms discussed in Sections 2 and 3.

Some versions of the network simplex method have been shown to solve the assignment problem in polynomial time. In particular, Orlin [81] shows that a natural version of the primal simplex method runs in polynomial time, and Balinski [6] gives a *signature method* that is a dual simplex algorithm for the assignment problem and runs in strongly polynomial time.

For discussion of parallel algorithms for the bipartite matching and assignment problems, see [10, 32, 42, 66, 76].

In this survey we would like to highlight the main ideas involved in designing highly efficient algorithms for network flow problems, rather than to discuss in detail the fastest algorithms. However, for easy reference, we summarize the running times of the fastest currently known algorithms in Table 1. For each problem we list all of the bounds that are best for some values of the parameters, but we only list the first algorithm achieving the same bound. Some of the bounds stated in

the table depend on the $O(m + n \log n)$ implementation of Dijkstra's shortest path algorithm [28].

Chapter 1

Preliminaries

In this chapter we define the problems addressed in this survey and review fundamental facts about these problems. These problems are the maximum flow problem, the minimum-cost circulation problem, the transshipment problem, and the generalized flow problem.

1.1 Flows and Residual Graphs

A *network* is a directed graph $G = (V, E)$ with a non-negative *capacity function* $u : E \to \mathbf{R}_\infty$.[1] We assume that G has no multiple arcs, *i.e.*, $E \subset V \times V$. If there is an arc from a vertex v to a vertex w, this arc is unique by the assumption, and we will denote it by (v, w). This assumption is for notational convenience only. We also assume, without loss of generality, that the input graph G is symmetric: $(v, w) \in E \iff (w, v) \in E$. A *flow network* is a network with two distinguished vertices, the *source* s and the *sink* t.

A *pseudoflow* is a function $f : E \to \mathbf{R}$ that satisfies the following constraints:

$$f(v, w) \leq u(v, w) \quad \forall (v, w) \in E \quad \text{(capacity constraint)}, \tag{1.1}$$

$$f(v, w) = -f(w, v) \quad \forall (v, w) \in E \quad \text{(flow antisymmetry constraint)}. \tag{1.2}$$

Remark: To gain intuition, it is often useful to think only about the non-negative components of a pseudoflow (or of a generalized pseudoflow, defined below). The antisymmetry constraints reflect the fact that a flow of value x from v to w can be thought of as a flow of value $(-x)$ from w to v. The negative flow values are introduced only for notational convenience. Note, for example, that one does not have to distinguish between lower and upper capacity bounds: the capacity of the arc (v, w) represents a lower bound on the flow value on the opposite arc.

[1] $\mathbf{R}_\infty = \mathbf{R} \bigcup \{\infty\}$.

Given a pseudoflow f, we define the *excess* function $e_f : V \to \mathbf{R}$ by $e_f(v) = \sum_{u \in V} f(u,v)$, the net flow into v. We will say that a vertex v has *excess* if $e_f(v)$ is positive, and has *deficit* if it is negative. For a vertex v, we define the *conservation constraint* by

$$e_f(v) = 0 \quad \text{(flow conservation constraint).} \tag{1.3}$$

Given a pseudoflow f, the *residual capacity* function $u_f : E \to \mathbf{R}$ is defined by $u_f(v,w) = u(v,w) - f(v,w)$. The *residual graph* with respect to a pseudoflow f is given by $G_f = (V, E_f)$, where $E_f = \{(v,w) \in E | u_f(v,w) > 0\}$.

1.2 The Maximum Flow Problem

To introduce the maximum flow problem, we need the following definitions in addition to the definitions of the previous section. Consider a flow network (G, u, s, t). A *preflow* is a pseudoflow f such that the excess function is non-negative for all vertices other than s and t. A *flow* f on G is a pseudoflow satisfying the conservation constraints for all vertices except s and t. The *value* $|f|$ of a flow f is the net flow into the sink $e_f(t)$. A *maximum flow* is a flow of maximum value (also called *an optimal flow*). The maximum flow problem is that of finding a maximum flow in a given flow network.

Given a flow f, we define an *augmenting path* to be a source-to-sink path in the residual graph. The following theorem, due to Ford and Fulkerson, gives an optimality criterion for maximum flows.

Theorem 1.2.1 [25] A flow is optimal if and only if its residual graph contains no augmenting path.

1.3 The Minimum-Cost Circulation Problem

A *circulation* is a pseudoflow with zero excess at every vertex. A *cost function* is a real-valued function on arcs $c : E \to \mathbf{R}$. Without loss of generality, we assume that costs are antisymmetric:

$$c(v,w) = -c(w,v) \quad \forall (v,w) \in E \quad \text{(cost antisymmetry constraint).} \tag{1.4}$$

The cost of a circulation f is given by

$$c(f) = \sum_{(v,w) \in E : f(v,w) \geq 0} f(v,w)c(v,w).$$

The *minimum-cost circulation* problem is that of finding a minimum-cost (*optimal*) circulation in an input network (G, u, c).

We have assumed that the capacities are non-negative. This assumption is no loss of generality. Given a circulation problem with some negative capacities one can find a feasible circulation f using one invocation of a maximum flow algorithm. (See [85], problem 11(e), p. 215.) The non-negative residual capacities $u(v,w) - f(v,w)$ then define an equivalent problem.

Next we state two criteria for the optimality of a circulation. Define the cost of a cycle Γ to be the sum of the costs of the arcs along the cycle and denote it by $c(\Gamma)$. The following theorem states the first optimality criterion.

Theorem 1.3.1 [15] A circulation is optimal if and only if its residual graph contains no negative-cost cycle.

To state the second criterion, we need the notions of a price function and a reduced cost function. A *price function* is a vertex labeling $p : V \to \mathbf{R}$. The *reduced cost function* with respect to a price function p is defined by $c_p(v,w) = c(v,w)+p(v)-p(w)$. These notions, which originate in the theory of linear programming, are crucial for many minimum-cost flow algorithms. As linear programming dual variables, vertex prices have a natural economic interpretation as the current market prices of the commodity. We can interpret the reduced cost $c_p(v,w)$ as the cost of buying a unit of commodity at v, transporting it to w, and then selling it. Due to the conservation constraints, the reduced costs define an equivalent problem.

Theorem 1.3.2 [25] A circulation f is optimal for the minimum-cost circulation problem (G,u,c) if and only if it is optimal for the problem (G,u,c_p) for every price function p.

The second optimality criterion is as follows.

Theorem 1.3.3 [25] A circulation f is optimal if and only if there is a price function p such that, for each arc (v,w),

$$c_p(v,w) < 0 \Rightarrow f(v,w) = u(v,w) \quad \text{(complementary slackness constraint).} \tag{1.5}$$

A minimum-cost circulation problem may have no optimal solution. The following theorem characterizes when this is the case.

Theorem 1.3.4 There exists a minimum-cost circulation if and only if the input network contains no negative-cost cycle consisting of arcs with infinite capacity.

Note that the maximum flow problem is a special case of the minimum-cost circulation problem (see e.g. [25]). To see this, consider a flow network and add a pair of new arcs (s,t) and (t,s) with $u(s,t) = 0$ and $u(t,s) = \infty$. Define the costs of all original arcs to be zero, and define $c(s,t) = -c(t,s) = 1$. A minimum-cost circulation in the resulting network restricted to the original network is a maximum flow in the original network.

1.4 The Transshipment Problem

In this section we define the (uncapacitated) *transshipment problem.* Although this problem is equivalent to the minimum-cost circulation problem, some algorithms are more natural in the context of the transshipment problem. For a more extensive discussion of this problem and the closely related transportation problem, see [72].

In the transshipment problem, all arc capacities are either zero or infinity. In addition, the input to the problem contains a demand function $d : V \to \mathbf{R}$ such that $\sum_{v \in V} d(v) = 0$. For the transshipment problem, the notion of the excess at a vertex is defined as follows:

$$e_f(v) = \sum_{w \in V} f(w, v) - d(v). \tag{1.6}$$

A pseudoflow f is *feasible* if the conservation constraints (1.3) hold for all vertices. The *transshipment problem* is that of finding a minimum-cost feasible pseudoflow in an input network. In the special case of integer demands, we shall use D to denote the maximum absolute value of a demand.

Theorems analogous to Theorem 1.3.1, 1.3.3 and 1.3.2. hold for the transshipment problem, and the analog of Theorem 1.3.4 holds for a transshipment problem that has a feasible pseudoflow.

We make two simplifying assumptions about the transshipment problem. First, we assume that, for the zero flow, all residual arcs have non-negative cost. This assumption can be validated by first checking whether the condition of Theorem 1.3.4 is satisfied (using a shortest path computation). In the case of a transshipment problem, the residual graph of the zero flow consists of the arcs with infinite capacity. If this graph has no negative-cost cycles, then define $p(v)$ to be the cost of a minimum-cost path from a fixed vertex s to the vertex v. The costs $c_p(v, w)$ define an equivalent problem that satisfies the assumption. Second, we assume that the residual graph of the zero flow is strongly connected. (There is a path between each pair of vertices.) This condition can be imposed by adding two new vertices x and y and adding an appropriately high-cost arc from x to y with $u(x, y) = \infty$ and $u(y, x) = 0$, and arcs between the new vertices and every other vertex in both directions such that $u(v, x) = u(y, v) = \infty$ and $u(x, v) = u(v, y) = 0$ for every vertex v. If the original problem has a feasible solution, then in every minimum-cost solution of the transformed problem all of the dummy arcs introduced to make the graph strongly connected have zero flow.

Next we show that the transshipment problem is equivalent to the minimum-cost circulation problem (see e.g. [25]). Given an instance of the transshipment problem, we construct an equivalent instance of the minimum-cost circulation problem as follows. Add two new vertices, x and y, and arcs (y, x) and (x, y) with $u(x, y) = 0$ and $u(y, x) = \sum_{v:d(v)>0} d(v)$. Define the cost of (y, x) to be small enough so that any simple cycle containing (y, x) has negative cost; for example, define $c(y, x) = -c(x, y) = -nC$. For every vertex $v \in V$ with $d(v) > 0$, add arcs (x, v) and (v, x) and define $u(x, v) = d(v)$, $u(v, x) = 0$, and $c(x, v) = c(v, x) = 0$. For every vertex $v \in V$ with $d(v) < 0$,

add arcs (v, y) and (y, v) and define $u(v, y) = -d(v)$, $u(y, v) = 0$, and $c(v, y) = c(y, v) = 0$. Consider an optimal solution f to the minimum-cost circulation problem in the resulting network. The transshipment problem is feasible if and only if f saturates all new arcs, and in this case the restriction of f to the original arcs gives an optimal solution to the transshipment problem.

Next we reduce the minimum-cost circulation problem to the transshipment problem. The reduction uses the technique of *edge-splitting*. Consider a minimum-cost circulation problem (G, u, c). First we make sure that for every edge $\{v, w\}$ of the network, either $u(v, w)$ or $u(w, v)$ is infinite. For every edge $\{v, w\}$ that does not satisfy the above condition, we introduce a new vertex $x_{(v,w)}$, and replace the arcs (v, w) and (w, v) by the arcs $(v, x_{(v,w)})$, $(x_{(v,w)}, w)$, $(w, x_{(v,w)})$, and $(x_{(v,w)}, v)$. Define the costs and capacities of the new arcs as follows:

$$\begin{array}{ll} u(v, x_{(v,w)}) = u(v, w), & c(v, x_{(v,w)}) = c(v, w), \\ u(x_{(v,w)}, w) = \infty, & c(x_{(v,w)}, w) = 0, \\ u(w, x_{(v,w)}) = u(w, v), & c(v, x_{(v,w)}) = 0, \\ u(x_{(v,w)}, v) = \infty, & c(x_{(v,w)}, v) = c(w, v). \end{array}$$

This defines an equivalent minimum-cost circulation problem in which the capacity of every edge is unbounded in at least one direction.

To complete the transformation, we need to force the additional restriction that every finite capacity is zero. Initialize the demand function d to be zero on all vertices. Let $\{v, w\}$ be an edge such that $u(w, v) = \infty$ and $u(v, w)$ is finite but not zero. Replacing $u(v, w)$ by zero, and $d(v)$ and $d(w)$ by $d(v) + u(v, w)$ and $d(w) - u(v, w)$, respectively, defines an equivalent problem with $u(v, w) = 0$.

Remark: This transformation increases the number of vertices: from a network with n vertices and m edges, we obtain a network with up to $n + m$ vertices and $2m$ edges. However, the resulting network has a special structure. It is important for the analyses in Chapter 5 to notice that the newly introduced vertices can be eliminated for shortest path computations. Consequently, this blowup in the number of vertices will not affect the running time of shortest path computations in the residual graph.

The *capacitated transshipment problem* generalizes both the minimum-cost circulation problem and the (uncapacitated) transshipment problem discussed above. It is the same as the uncapacitated transshipment problem without the assumption that all arc capacities are infinite or zero. The reductions above can be extended to show that the capacitated transshipment problem is equivalent to the minimum-cost circulation problem. Whereas the previous simpler formulations are better suited for designing algorithms, the more general form can be useful in applications.

1.5 The Generalized Flow Problem

In this section we define generalized pseudoflows, generalized flows, and the generalized flow problem, and compare these concepts with their counterparts discussed in the previous sections. For alternative formulations of the problem, see e.g. [41, 72]. (This problem is also known as the problem of flows with losses and gains.)

The generalized flow problem is given by a network with a distinguished vertex, the *source* s, and a *gain function* $\gamma : E \to \mathbf{R}^{+}$[2] on the arcs. We assume (without loss of generality) that the gain function is antisymmetric:

$$\gamma(v,w) = \frac{1}{\gamma(w,v)} \quad \forall (v,w) \in E \quad \text{(gain antisymmetry constraints).} \tag{1.7}$$

In the case of ordinary flows, if $f(v,w)$ units of flow are shipped from v to w, $f(v,w)$ units arrive at w. In the case of generalized flows, if $g(v,w)$ units of flow are shipped from v to w, $\gamma(v,w)g(v,w)$ units arrive at w. A *generalized pseudoflow* is a function $g : E \to \mathbf{R}$ that satisfies the capacity constraints (1.1) and the generalized antisymmetry constraints:

$$g(v,w) = -\gamma(w,v)g(w,v) \quad \forall (v,w) \in E \quad \text{(generalized antisymmetry constraints).} \tag{1.8}$$

The gain of a path (cycle) is the product of the gains of the arcs on the path (cycle). For a given generalized pseudoflow g, the residual capacity and the residual graph $G_g = (V, E_g)$ are defined in the same way as for pseudoflows. The *excess* $e_g(v)$ of a generalized pseudoflow g at a vertex v is defined by

$$e_g(v) = -\sum_{(v,w)\in E} g(v,w). \tag{1.9}$$

We will say that a vertex v has *excess* if $e_g(v)$ is positive and *deficit* if it is negative.

A *generalized flow* is a generalized pseudoflow that satisfies the conservation constraints (1.3) at all vertices except at the source. The *value* of a generalized pseudoflow g is the excess $e_g(s)$. The generalized flow problem is that of finding a generalized flow of maximum value in a given network. In our discussion of this problem, we shall assume that the capacities are integers, and that each gain is a rational number given as the ratio of two integers. Let B denote the largest integer used in the specification of the capacities and gains.

A *flow-generating cycle* is a cycle whose gain is greater than 1, and a *flow-absorbing cycle* is a cycle whose gain is less than 1. Observe that if one unit of flow leaves a vertex v and travels along a flow-generating cycle, more than one unit of flow arrives at v. Thus we can *augment* the flow

[2] $\mathbf{R}^{+}$ denotes the set of positive reals.

along this cycle; that is, we can increase the excess at any vertex of the cycle while preserving the excesses at the other vertices, by increasing the flow along the arcs in the cycle (and correspondingly decreasing the flow on the opposite arcs to preserve the generalized antisymmetry constraints).

The generalized flow problem has the following interpretation in financial analysis. The commodity being moved is money, nodes correspond to different currencies and securities, arcs correspond to possible transactions, and gain factors represent the prices or the exchange rates (see Section 6.1). From the investor's point of view, a residual flow-generating cycle is an opportunity to make a profit. It is possible to take advantage of this opportunity, however, only if there is a way to transfer the profit to the investor's bank account (the source vertex). This motivates the following definition. A *generalized augmenting path (GAP)* is a residual flow-generating cycle and a (possibly trivial) residual path from a vertex on the cycle to the source. Given a generalized flow and a GAP in the residual graph, we can augment the flow along the GAP, increasing the value of the current flow. The role of GAP's in the generalized flow problem is similar to the role of negative-cost cycles in the minimum-cost circulation problem – both can be used to augment the flow and improve the value of the current solution. Onaga [78] proved that the non-existence of GAP's in the residual graph characterizes the optimality of a generalized flow.

Theorem 1.5.1 [78] **A generalized flow is optimal if and only if its residual graph contains no GAP.**

Using the linear programming dual of the problem, it is possible to give an alternate optimality criterion, similar to the criterion in Theorem 1.3.3 for the minimum-cost circulation problem. A *price function* p is a labeling of the vertices by real numbers, such that $p(s) = 1$. As in the case of the minimum-cost circulation problem, vertex prices can be interpreted as market prices of the commodity at vertices. If a unit of flow is purchased at v and shipped to w, then $\gamma(v, w)$ units arrive at w. Buying the unit at v costs $p(v)$, and selling $\gamma(v, w)$ units at w returns $p(w)\gamma(v, w)$. Thus the reduced cost of (v, w) is defined as

$$c_p(v, w) = p(v) - p(w)\gamma(v, w).$$

Linear programming duality theory provides the following optimality criterion.

Theorem 1.5.2 [60] **A generalized flow is optimal if and only if there exists a price function p such that the complementary slackness conditions (1.5) hold for each arc $(v, w) \in E$.**

One generalization of the problem, namely the *generalized flow problem with costs*, is worth mentioning here. As the name suggests, in this problem each arc (v, w) has a cost $c(v, w)$ in addition to its gain. The goal is to find a generalized flow that maximizes $e_g(s) - c(g)$, where $c(g) = \sum_{(v,w):g(v,w)>0} c(v, w)g(v, w)$. The introduction of costs enriches the combinatorial structure of the problem and allows the modeling of more complex problems, in particular economic processes. For

example, a positive cost flow-generating cycle with a path leading to a negative cost flow-absorbing cycle may be used as a producer-consumer model. The generalized flow problem with costs has not been studied as much as the other problems discussed in this survey, and the only polynomial-time algorithms known for this problem are based on general linear programming methods [63, 101].

1.6 The Restricted Problem

Next we introduce a special variant of the generalized flow problem, and show that this variant is equivalent to the original problem.

Consider for a moment the following variation of the generalized flow problem: given a flow network with a source $s \in V$ and a sink $t \in V$, find a generalized pseudoflow with maximum excess at t and zero excess at each vertex other than s and t. Onaga [77] suggested the study of the special case of this problem in which the residual graph of the zero flow has no flow-generating cycles. We shall consider the corresponding special case of the generalized flow problem in which the residual graph of the zero flow has the property that all flow-generating cycles pass through the source. (If there are no flow-generating cycles, the zero flow is the optimal.) We shall also assume that the residual graph of the zero flow is strongly connected. A generalized flow network in which the residual graph of the zero flow is strongly connected and which has no flow-generating cycles not passing through the source is called a *restricted network*. The *restricted problem* is the generalized flow problem on a restricted network. The restricted problem has a simpler combinatorial structure that leads to simpler algorithms. Moreover, it turns out that the restricted problem is equivalent to the generalized flow problem. All of the algorithms that we review solve the restricted problem. In the rest of this section we shall review basic properties of the restricted problem, and we outline the reduction. In Chapter 6 we will use the term "generalized flow problem" to mean the restricted problem.

One of the nice facts about the restricted problem is that the optimality condition given by Theorem 1.5.1 simplifies in this case, and becomes very similar to Theorem 1.3.1. This characterization, which is also due to Onaga [77], can be deduced from Theorem 1.5.1 with the use of the following lemma. The lemma can be proved by induction on the number of arcs with positive flow.

Lemma 1.6.1 Let g be a generalized pseudoflow in a restricted network. If the excess at every vertex is non-negative, then for every vertex v there exists a path from v to s in the residual graph G_g.

Theorem 1.6.2 [78] A generalized flow g in a restricted network is optimal if and only if the residual graph of g contains no flow-generating cycles.

Note that the condition in the theorem can be formulated equivalently as "if and only if the residual graph of g has no negative-cost cycles, where the cost function is given by $c = -\log \gamma$."

Theorem 1.6.3 [41] The generalized flow problem can be reduced to the restricted problem in $O(nm)$ time.

Proof sketch: Given an instance of the generalized flow problem, reduce it to an instance in which all the flow-generating cycles pass through s, as follows. Let h be the generalized pseudoflow that saturates all arcs of gain greater than 1 and is zero on all arcs of gain 1. Define a new generalized flow network containing each arc (v, w) of the original network, with new capacity $u(v, w) - h(v, w)$. For each vertex $v \in V$ such that $e_h(v) > 0$, add an arc (s, v) of capacity $u(s, v) = e_h(v)/\alpha$, and with gain $\gamma(s, v) = \alpha$, where $\alpha = B^n$. Also add a reverse arc (v, s) with $u(v, s) = 0$. For each vertex $v \in V$ with $e_h(v) < 0$, add an arc (v, s) of capacity $u(v, s) = e_h(v)$ and with gain $\gamma(v, s) = \alpha$; also add a reverse arc (s, v) with $u(v, s) = 0$. Let $\hat{g}$ be an optimal generalized flow in the new network. Consider the generalized pseudoflow $\hat{g} + h$ restricted to the arcs of the original network. Using the Decomposition Theorem 1.7.3 below, one can show that the value of this generalized pseudoflow is equal to the value of the optimal generalized flow, and an optimal generalized flow can be constructed from this generalized pseudoflow in $O(nm)$ time. Intuitively, the new arcs ensure that vertices having excesses with respect to h are supplied with an adequate amount of "almost free" flow, and vertices having deficits with respect to h can deliver the corresponding amount of flow very efficiently to the source.

Having eliminated flow-generating cycles not containing the source, we can impose the strong connectivity condition by deleting all of the graph except the strongly connected component containing the source. This does not affect the value of an optimum solution by Theorem 1.5.1. ∎

Note that this transformation increases the size of the numbers in the problem description. However, the polynomial-time algorithms described later depend only on T ($\leq B^n$), the maximum product of gains along a simple path, and L ($\leq B^{2m}$), the product of denominators of arc gains; these parameters do not increase significantly.

1.7 Decomposition Theorems

A useful property of flows and generalized flows is the fact that they can be decomposed into a small number of "primitive elements". These elements depend on the problem under consideration. In this section we review decomposition theorems for circulations, pseudoflows, and generalized pseudoflows.

In the case of circulations, the primitive elements of the decomposition are flows along simple cycles.

Theorem 1.7.1 [25] For every circulation f, there exists a collection of $k \leq m$ circulations $g_1, \ldots, g_k$ such that for every i, the graph induced by the set of arcs on which g_i is positive is a simple cycle consisting of arcs (v, w) with $f(v, w) > 0$, and

$$f(v,w) = \sum_i g_i. \tag{1.10}$$

Such a decomposition can be found in $O(nm)$ time.

For pseudoflows (or flows), the primitive elements of the decomposition are flows along simple cycles and flows along simple paths.

Theorem 1.7.2 [25] For every pseudoflow f, there exists a collection of $k \leq m$ pseudoflows $g_1, \ldots, g_k$, such that for every i, the graph induced by the set of arcs on which g_i is positive is either a simple cycle or a simple path from a vertex with deficit to a vertex with excess; it consists of arcs (v,w) with $f(v,w) > 0$, and (1.10) is satisfied. Such a decomposition can be found in $O(nm)$ time.

The five primitive elements of the decomposition for generalized pseudoflows g are defined as follows. The set of arcs on which an element of the decomposition of g is positive is a subset of the arcs on which g is positive. Let $T(g)$ denote the set of vertices with excesses and let $S(g)$ denote the set of vertices with deficits. The five types of primitive elements are classified according to the graph induced by the set of arcs with positive flow. *Type I*: A path from a vertex in $S(g)$ to a vertex in $T(g)$. Such a flow creates a deficit and an excess at the two ends of the path. *Type II*: A flow-generating cycle and a path leading from this cycle to a vertex in $T(g)$. Such a flow creates excess at the end of the path. (If the path ends at the source, then this corresponds to a GAP.) *Type III*: A flow-absorbing cycle and a path from a vertex in $S(g)$ to this cycle. Such a flow creates a deficit at the end of the path. *Type IV*: A cycle with unit gain. Such a flow does not create any excesses or deficits. *Type V*: A pair of cycles connected by a path, where one of the cycles generates flow and the other one absorbs it. Such a flow does not create any excesses or deficits.

Theorem 1.7.3 [41, 51] For a generalized pseudoflow g there exist $k \leq m$ primitive pseudoflows $g_1, \ldots, g_k$ such that, for each i, g_i is of one of the five types described above, and (1.10) is satisfied. Such a decomposition can be found in $O(nm)$ time.

Remark: In all three of these theorems the time to find the claimed decomposition can be reduced to $O(m \log n)$ by using the dynamic tree data structure that will be discussed in Section 2.5.

Chapter 2

The Maximum Flow Problem

2.1 Introduction

The maximum flow problem has been studied for over thirty years. The classical methods for solving this problem are the Ford-Fulkerson *augmenting path method* [26, 25], the closely related *blocking flow method* of Dinic [20], and an appropriate variant of the *network simplex method* (see e.g. [59]). A *push-relabel method* for solving maximum flow problems has been recently proposed by Goldberg [38] and fully developed by Goldberg and Tarjan [43, 44]. Recently, parallel and distributed algorithms for the maximum flow problem have been studied as well.

Many maximum flow algorithms use scaling techniques and data structures to achieve better running time bounds. The scaling techniques used by maximum flow algorithms are *capacity scaling*, introduced by Edmonds and Karp [21] in the context of the minimum-cost circulation problem, and the closely related *excess scaling* of Ahuja and Orlin [3]. The dynamic tree data structure of Sleator and Tarjan [92, 93] makes it possible to speed up many maximum flow algorithms.

The technical part of this survey deals with the push-relabel method, which gives better sequential and parallel complexity bounds than previously known methods, and seems to outperform them in practice. In addition to describing the basic method, we show how to use excess scaling and dynamic trees to obtain faster implementations of the method. In this section we discuss the previous approaches and their relationship to the push-relabel method.

The augmenting path algorithm of Ford and Fulkerson is based on the fact that a flow is maximum if and only if there is no *augmenting path*. The algorithm repeatedly finds an augmenting path and augments along it, until no augmenting path exists. Although this simple generic method need not terminate if the network capacities are reals, and it can run in exponential time if the capacities are integers represented in binary [25], it can be made efficient by restricting the choice of augmenting paths. Edmonds and Karp [21] have shown that two strategies for selecting the next augmenting path give polynomial time bounds. The first such strategy is to select, at each iteration, a shortest augmenting path, where the length of a path is the number of arcs on the path.

The second strategy is to select, at each iteration, a fattest path, where the fatness of a path is the minimum of the residual capacities of the arcs on the path.

Independently, Dinic [20] suggested finding augmenting paths in phases, handling all augmenting paths of a given shortest length in one phase by finding a *blocking flow* in a layered network. The shortest augmenting path length increases from phase to phase, so that the blocking flow method terminates in $n-1$ phases. Dinic's discovery motivated a number of algorithms for the blocking flow problem. Dinic proposed an $O(nm)$ blocking flow algorithm. Soon thereafter, Karzanov [67] proposed an $O(n^2)$ algorithm, which achieves the best known bound for dense graphs. Karzanov also introduced the concept of a preflow; this concept is used by many maximum flow algorithms. Simpler algorithms achieving an $O(n^2)$ bound are described in [73, 95]. A sequence of algorithms achieving better and better running times on non-dense graphs has been proposed [17, 31, 34, 35, 47, 90, 93]. The algorithm of [31] uses capacity scaling; the algorithms of [93] and [47] use dynamic trees. The fastest currently-known sequential algorithm for the blocking flow problem, due to Goldberg and Tarjan, runs in $O(m\log(n^2/m))$ time [47].

The push-relabel method [38, 39, 43, 47] replaces the layered network of the blocking flow method by a distance labeling of the vertices, which gives an estimate of the distance to the sink in the residual graph. The algorithm maintains a preflow and a distance labeling, and uses two simple operations, *pushing* and *relabeling*, to update the preflow and the labeling, repeating them until a maximum flow is found. The *pushing* operation is implicit in Karzanov's algorithm. The construction of a layered network at each iteration of Dinic's method can be viewed as a sequence of *relabeling* operations. Unlike the blocking flow method, the push-relabel method solves the maximum flow problem directly, without reducing the problem to a sequence of blocking flow subproblems. As a result, this method is more general and flexible than the blocking flow method, and leads to improved sequential and parallel algorithms. Ahuja and Orlin [3] suggested one way of recasting algorithms based on Dinic's approach into the push-relabel framework.

Goldberg's *FIFO*[1] algorithm [38] runs in $O(n^3)$ time. Goldberg and Tarjan [43, 44] showed how to use the dynamic tree data structure to improve the running time of this algorithm to $O(nm\log(n^2/m))$. They also gave a *largest-label* variant of the algorithm, for which they obtained the same time bounds – $O(n^3)$ without dynamic trees and $O(nm\log(n^2/m))$ with such trees. Cheriyan and Maheshwari [16] showed that (without dynamic trees) the FIFO algorithm runs in $\Omega(n^3)$ time, whereas the largest-label algorithm runs in $O(n^2\sqrt{m})$ time. Ahuja and Orlin [3] described an $O(nm+n^2\log U)$-time version of the push-relabel method based on *excess scaling*. Ahuja, Orlin, and Tarjan [4] gave a modification of the Ahuja-Orlin algorithm that runs in $O(nm+n^2\sqrt{\log U})$ time without the use of dynamic trees and in $O(nm\log(\frac{n}{m}\sqrt{\log U}+2))$ time with them.

The primal network simplex method (see e.g. [59]) is another classical method for solving the maximum flow problem. Cunningham [18] gave a simple pivoting rule that avoids cycling due to

[1] FIFO is an abbreviation of *first-in, first-out*.

degeneracy. Of course, if a simplex algorithm does not cycle, it must terminate in exponential time. Until recently, no polynomial time pivoting rule was known for the primal network simplex method. Goldfarb and Hao [50] have given such a rule. The resulting algorithm does $O(nm)$ pivots, which correspond to $O(nm)$ flow augmentations; it runs in $O(n^2m)$ time. Interestingly, the blocking flow method also makes $O(nm)$ flow augmentations. Unlike the blocking flow method, the Goldfarb-Hao algorithm does not necessarily augment along shortest augmenting paths. In their analysis, Goldfarb and Hao use a variant of distance labeling and a variant of the *relabeling* operation mentioned above. Dynamic trees can be used to obtain an $O(nm \log n)$-time implementation of their algorithm [40].

The best currently known sequential bounds for the maximal flow problem are $O(nm \log(n^2/m))$ and $O(nm \log(\frac{n}{m}\sqrt{\log U}+2))$. Note that, although the running times of the known algorithms come very close to an $O(mn)$ bound, the existence of a maximum flow algorithm that meets this bound remains open.

With the increasing interest in parallel computing, parallel and distributed algorithms for the maximum flow problem have received a great deal of attention. The first parallel algorithm for the problem, due to Shiloach and Vishkin [91], runs in $O(n^2 \log n)$ time on an n-processor PRAM [27] and uses $O(n)$ memory per processor. In a synchronous distributed model of computation, this algorithm runs in $O(n^2)$ time using $O(n^3)$ messages and $O(n)$ memory per processor. The algorithm of Goldberg [38, 39, 44] uses less memory than that of Shiloach and Vishkin: $O(m)$ memory for the PRAM implementation and $O(\Delta)$ memory per processor for the distributed implementation (where Δ is the processor degree in the network). The time, processor, and message bounds of this algorithm are the same as those of the Shiloach-Vishkin algorithm. Ahuja and Orlin [3] developed a PRAM implementation of their excess-scaling algorithm. The resource bounds are $\lceil m/n \rceil$ processors, $O(m)$ memory, and $O(n^2 \log n \log U)$ time. Cheriyan and Maheshwari [16] proposed a synchronous distributed implementation of the largest-label algorithm that runs in $O(n^2)$ time using $O(\Delta)$ memory per processor and $O(n^2\sqrt{m})$ messages.

For a long time, the primal network simplex method was the method of choice in practice. A study of Goldfarb and Grigoriadis [49] suggested that the algorithm of Dinic [20] performs better than the network simplex method and better than the later algorithms based on the blocking flow method. Recent studies of Ahuja and Orlin (personal communication) and Grigoriadis (personal communication) show superiority of various versions of the push-relabel method to Dinic's algorithm. An experimental study of Goldberg [39] shows that a substantial speedup can be achieved by implementing the FIFO algorithm on a highly parallel computer.

More efficient algorithms have been developed for the special case of planar networks. Ford and Fulkerson [26] have observed that the the maximum flow problem on a planar network is related to a shortest path problem on the planar dual of the network. The algorithms in [26, 8, 54, 55, 58,

```
procedure generic (V, E, u);
    [initialization]
    ∀(v, w) ∈ E do begin
        f(v, w) ← 0;
        if v = s then f(s, w) ← u(s, w);
        if w = s then f(v, s) ← −u(s, v);
    end;
    ∀w ∈ V do begin
        e_f(w) ← Σ_{(v,w)∈E} f(v, w);
        if w = s then d(w) = n else d(w) = 0;
    end;
    [loop]
    while ∃ an active vertex do
        select an update operation and apply it;
    return(f);
end.
```

Figure 2.1: The generic maximum flow algorithm.

62, 74, 86] make clever use of this observation.

2.2 A Generic Algorithm

In this section we describe the generic push-relabel algorithm [39, 43, 44]. First, however, we need the following definition. For a given preflow f, a *distance labeling* is a function d from the vertices to the non-negative integers such that $d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for all residual arcs (v, w). The intuition behind this definition is as follows. Define a *distance graph* G_f^* as follows. Add an arc (s, t) to G_f. Define the length of all residual arcs to be equal to one and the length of the arc (s, t) to be n. Then d is a "locally consistent" estimate on the distance to the sink in the distance graph. (In fact, it is easy to show that d is a lower bound on the distance to the sink.) We denote by $d_{G_f^*}(v, w)$ the distance from vertex v to vertex w in the distance graph. The generic algorithm maintains a preflow f and a distance labeling d for f, and updates f and d using *push* and *relabel* operations. To describe these operations, we need the following definitions. We say that a vertex v is *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$. Note that a preflow f is a flow if and only if there are no active vertices. An arc (v, w) is *admissible* if $(v, w) \in E_f$ and $d(v) = d(w) + 1$.

The algorithm begins with the preflow f that is equal to the arc capacity on each arc leaving the source and zero on all arcs not incident to the source, and with some initial labeling d. The algorithm then repetitively performs, in any order, the *update operations*, *push* and *relabel*, described in Figure 2.2. When there are no active vertices, the algorithm terminates. A summary of the algorithm appears in Figure 2.1.

push(v, w).
Applicability: v is active **and** (v, w) is admissible.
Action: send $\delta \in (0, \min(e_f(v), u_f(v, w))]$ units of flow from v to w.

relabel(v).
Applicability: either s or t is reachable from v in G_f **and** $\forall w \in V\ u_f(v, w) = 0$ or $d(w) \geq d(v)$.
Action: replace $d(v)$ by $\min_{(v,w) \in E_f} \{d(w)\} + 1$.

Figure 2.2: The update operations. The *pushing* operation updates the preflow, and the *relabeling* operation updates the distance labeling. Except for the excess scaling algorithm, all algorithms discussed in this section push the maximum possible amount δ when doing a push.

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

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 $d(v) = d_{G_f^*}(v, t)$ for $v \in V$, where f is the initial preflow. The latter labeling can be computed in $O(m)$ time using backwards breadth-first searches from the sink and from the source in the residual graph. The resource bounds we shall 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. In practice, it is preferable to start with the most accurate values of the distance labels, and to update the distance labels periodically by using backward breadth-first search.

Remark: By giving priority to relabeling operations, it is possible to maintain the following invariant: Just before a push, d gives the exact distance to t in the distance graph. Furthermore, it is possible to implement the relabeling operations so that the total work done to maintain the distance labels is $O(nm)$ (see e.g. [44]). Since the running time bounds derived in this section are $\Omega(nm)$, one can assume that the relabeling is done in this way. In practice, however, maintaining exact distances is expensive; a better solution is to maintain a valid distance labeling and periodically update it to the exact labeling.

Next we turn our attention to the correctness and termination of the algorithm. Our proof of correctness is based on Theorem 1.2.1. The following lemma is important in the analysis of the algorithm.

Lemma 2.2.1 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.

Using this lemma and induction on the number of update operations, it can be shown that

one of the two update operations must be applicable to an active vertex, and that the operations maintain a valid distance labeling and preflow.

Theorem 2.2.2 [44] Suppose that the algorithm terminates. Then the preflow f is a maximum flow.

Proof: When the algorithm terminates, all vertices in $V - \{s, t\}$ must have zero excess, because there are no active vertices. Therefore f must be a flow. We show that if f is a preflow and d is a valid labeling for f, then the sink t is not reachable from the source s in the residual graph G_f. Then Theorem 1.2.1 implies that the algorithm terminates with a maximum flow.

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 $d(s) = n$. ∎

The key to the running time analysis of the algorithm is the following lemma, which shows that distance labels cannot increase too much.

Lemma 2.2.3 At any time during the execution of the algorithm, 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 vertex labels by only by means of the *relabeling* operation, it is enough to prove the lemma for a vertex v such that s or t is reachable from v in G_f. Thus there is a simple path from v to s or t in G_f. Let $v = v_0, v_1, \ldots, v_l$ 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, since $d(v_l)$ is either n or 0, we have $d(v) = d(v_0) \leq d(v_l) + l \leq n + (n - 1) = 2n - 1$. ∎

Lemma 2.2.3 limits the number of relabeling operations, and 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 2.2.4 The number of relabeling operations is at most $2n - 1$ per vertex and at most $(2n - 1)(n - 2) < 2n^2$ overall.

Lemma 2.2.5 The number of saturating pushes is at most nm.

Proof: For an arc $(v, w) \in E$, consider the saturating pushes from v to w. After one such push, $u_f(v, w) = 0$, and another such push cannot occur until $d(w)$ increases by at least 2, a push from w to v occurs, and $d(v)$ increases by at least 2. If we charge each saturating push from v to w except the first to the preceding label increase of v, we obtain an upper bound of n on the number of such pushes. ∎

The most interesting part of the analysis is obtaining a bound on the number of nonsaturating pushes. For this we use amortized analysis and in particular the *potential function* technique (see e.g. [96]).

Lemma 2.2.6 The number of nonsaturating pushing operations is at most $2n^2m$.

Proof: We define the *potential* Φ of the current preflow f and labeling d by the formula $\Phi = \sum_{\{v|v \text{ is active}\}} d(v)$. We have $0 \leq \Phi \leq 2n^2$ by Lemma 2.2.3. Each nonsaturating push, say from a vertex v to a vertex w, decreases Φ by at least one, since $d(w) = d(v) - 1$ and the push makes v inactive. It follows that the total number of nonsaturating pushes over the entire algorithm is at most the sum of the increases in Φ during the course of the algorithm, since $\Phi = 0$ both at the beginning and at the end of the computation. Increasing the label of a vertex v by an amount k increases Φ by k. The total of such increases over the algorithm is at most $2n^2$. A saturating push can increase Φ by at most $2n - 2$. The total of such increases over the entire algorithm is at most $(2n-2)nm$. Summing gives a bound of at most $2n^2 + (2n-2)nm \leq 2n^2m$ on the number of nonsaturating pushes. ∎

Theorem 2.2.7 [44] The generic algorithm terminates after $O(n^2m)$ update operations.

Proof: Immediate from Lemmas 2.2.4, 2.2.5, and 2.2.6. ∎

The running time of the generic algorithm depends upon the order in which update operations are applied and on implementation details. In the next sections we explore these issues. First we give a simple implementation of the generic algorithm in which the time required for the nonsaturating pushes dominates the overall running time. Sections 2.3 and 2.4 specify orders of the update operations that decrease the number of nonsaturating pushes and permit $O(n^3)$ and $O(mn + n^2 \log U)$-time implementations. Section 2.5 explores an orthogonal approach. It shows how to use sophisticated data structures to reduce the time per nonsaturating push rather than the number of such pushes.

2.3 Efficient Implementation

Our first step toward an efficient implementation is a way of combining the update operations locally. We need some data structures to represent the network and the preflow. We call an unordered pair $\{v, w\}$ such that $(v, w) \in E$ an *edge* of G. We associate the three values $u(v, w)$, $u(w, v)$, and $f(v, w)(= -f(w, v))$ with each edge $\{v, w\}$. Each vertex v has a list of the incident 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

```
discharge(v).
Applicability: v is active.
Action:        let {v, w} be the current edge of v;
               time-to-relabel ← false;
               repeat
                  if (v, w) is admissible then push(v, w)
                  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 list
                     else begin
                        make the first edge on the edge list of v the current edge;
                        time-to-relabel ← true;
                     end;
               until e_f(v) = 0 or time-to-relabel;
               if time-to-relabel then relabel(v);
```

Figure 2.3: The discharge operation.

for a pushing operation from v. Initially, the current edge of v is the first edge on the edge list of v. The main loop of the implementation consists of repeating the *discharge* operation described in Figure 2.3 until there are no active vertices. (We shall discuss the maintenance of active vertices later.) The *discharge* operation is applicable to an active vertex v. This operation iteratively attempts to push the excess at v through the current edge $\{v, w\}$ of v if a pushing operation is applicable to this edge. If not, the operation replaces $\{v, w\}$ as the current edge of v by the next edge on the edge list of v; or, if $\{v, w\}$ is the last edge on this list, it makes the first edge on the list the current one and relabels v. The operation stops when the excess at v is reduced to zero or v is relabeled.

The following lemma shows that *discharge* does relabeling correctly; the proof of the lemma is straightforward.

Lemma 2.3.1 The discharge operation does a relabeling only when the relabeling operation is applicable.

Lemma 2.3.2 The version of the generic push/relabel algorithm based on discharging runs in $O(nm)$ time plus the total time needed to do the nonsaturating pushes and to maintain the set of active vertices.

Any representation of the set of active vertices that allows insertion, deletion, and access to some active vertex in $O(1)$ time results in an $O(n^2m)$ running time for the discharge-based algorithm, by Lemmas 2.2.6 and 2.3.2. (Pushes can be implemented in $O(1)$ time per push.)

By processing active vertices in a more restricted order, we obtain improved performance. Two natural orders were suggested in [43, 44]. One, the *FIFO algorithm*, is to maintain the set of active vertices as a queue, always selecting for discharging the front vertex on the queue and adding newly

```
procedure process-vertex;
    remove a vertex v from B_b;
    old-label ← d(v);
    discharge(v);
    add each vertex w made active by the discharge to B_d(w);
    if d(v) ≠ old-label then begin
        b ← d(v);
        add v to B_b;
    end
    else if B_b = ∅ then b ← b − 1;
end.
```

Figure 2.4: The *process-vertex* procedure.

active vertices to the rear of the queue. The other, the *largest-label algorithm*, is to always select for discharging a vertex with the largest label. The FIFO algorithm runs in $O(n^3)$ time [43, 44] and the largest-label algorithm runs in $O(n^2\sqrt{m})$ time [16]. We shall derive an $O(n^3)$ time bound for both algorithms, after first describing in a little more detail how to implement largest-label selection.

The implementation maintains an array of sets B_i, $0 \leq i \leq 2n-1$, and an index b into the array. Set B_i consists of all active vertices with label i, represented as a doubly-linked list, so that insertion and deletion take $O(1)$ time. The index b is the largest label of an active vertex. During the initialization, when the arcs going out of the source are saturated, the resulting active vertices are placed in B_0, and b is set to 0. At each iteration, the algorithm removes a vertex from B_b, processes it using the *discharge* operation, and updates b. The algorithm terminates when b becomes negative, *i.e.*, when there are no active vertices. This processing of vertices, which implements the *while* loop of the generic algorithm, is described in Figure 2.4.

To understand why the *process-vertex* procedure correctly maintains b, note that *discharge*(v) either relabels v or gets rid of all excess at v, but not both. In the former case, v is the active vertex with the largest distance label, so b must be increased to $d(v)$. In the latter case, the excess at v has been moved to vertices with distance labels of $b-1$, so if B_b is empty, then b must be decreased by one. The total time spent updating b during the course of the algorithm is $O(n^2)$.

The bottleneck in both the FIFO method and the largest-label method is the number of non-saturating pushes. We shall obtain an $O(n^3)$ bound on the number of such pushes by dividing the computation into *phases*, defined somewhat differently for each method. For the FIFO method, phase 1 consists of the discharge operations applied to the vertices added to the queue by the initialization of f; phase $i+1$, for $i \leq 1$, consists of the discharge operations applied to the vertices added to the queue during phase i. For the largest-label method, a phase consists of a maximal interval of time during which b remains constant.

Lemma 2.3.3 The number of phases during the running of either the FIFO or the largest-label algo-

rithm is at most $4n^2$.

Proof: Define the potential Φ of the current f and d by $\Phi = \max_{\{v|v \text{ is active}\}} d(v)$, with the maximum taken to be zero if there are no active vertices. (In the case of the largest-label algorithm, $\Phi = b$ except on termination.) There can be only $2n^2$ phases that do one or more relabelings. A phase that does no relabeling decreases Φ by at least one. The initial and final values of Φ are zero. Thus the number of phases that do no relabeling is at most the sum of the increases in Φ during the computation. The only increases in Φ are due to label increases; an increase of a label by k can cause Φ to increase by up to k. Thus the sum of the increases in Φ over the computation is at most $2n^2$, and so is the number of phases that do no relabeling. ∎

Theorem 2.3.4 [44] Both the FIFO and the largest-label algorithm run in $O(n^3)$ time.

Proof: For either algorithm, there is at most one nonsaturating push per vertex per phase. Thus by Lemma 2.3.3 the total number of nonsaturating pushes is $O(n^3)$, as is the running time by Lemma 2.3.2. ∎

Cheriyan and Maheshwari [16], by means of an elegant balancing argument, were able to improve the bound on the number of nonsaturating pushes in the largest-label algorithm to $O(n^2\sqrt{m})$, giving the following result:

Theorem 2.3.5 [16] The largest-label algorithm runs in $O(n^2\sqrt{m})$ time.

2.4 Excess Scaling

A different approach to active vertex selection leads to running time bounds dependent on the size U of the largest capacity as well as on the graph size. This approach, *excess scaling*, was introduced by Ahuja and Orlin [3] and developed further by Ahuja, Orlin, and Tarjan [4]. We shall describe in detail a slight revision of the original excess-scaling algorithm, which has a running time of $O(nm + n^2 \log U)$.

For the termination of the excess-scaling method, all arc capacities must be integral; hence we assume throughout this section that this is the case. The method preserves integrality of the flow throughout the computation. It depends on a parameter Δ that is an upper bound on the maximum excess of an active vertex. Initially $\Delta = 2^{\lceil \log U \rceil}$. The algorithm proceeds in phases; after each phase, Δ is halved. When $\Delta < 1$, all active vertex excesses must be zero, and the algorithm terminates. Thus the algorithm terminates after at most $\log_2 U + 1$ phases. To maintain the invariant that no active vertex excess exceeds Δ, the algorithm does not always push the maximum possible amount when doing a pushing operation. Specifically, when pushing from a vertex v to a

vertex w, the algorithm moves an amount of flow δ given by $\delta = \min\{e_f(v),\, u_f(v,w),\, \Delta - e_f(w)\}$ if $w \neq t, \delta = \min\{e_f(v),\, u_f(v,w)\}$ if $w = t$. That is, δ is the maximum amount that can be pushed while maintaining the invariant.

The algorithm consists of initialization of the preflow, the distance labels, and Δ, followed by a sequence of *process-vertex* operations of the kind described in Section 2.3. Vertex selection for *process-vertex* operations is done by the *large excess, smallest label rule*: process an active vertex v with $e_f(v) > \Delta/2$; among all candidate vertices, choose one of smallest label. When every active vertex v has $e_f(v) \leq \Delta/2$, Δ is halved and a new phase begins; when there are no active vertices, the algorithm stops.

Since the excess-scaling algorithm is a version of the generic *process-vertex*-based algorithm described in Section 2.3, Lemma 2.3.2 applies. The following lemma bounds the number of nonsaturating pushes:

Lemma 2.4.1 The number of nonsaturating pushes during the excess-scaling algorithm is $O(n^2 \log U)$.

Proof: We define a potential Φ by $\Phi = \sum_{\{v|v \text{ is active}\}} e_f(v)\, d(v)/\Delta$. Since $e_f(v)/\Delta \leq 1$ for any active vertex, $0 \leq \Phi \leq 2n^2$. Every pushing operation decreases Φ. Consider a nonsaturating push from a vertex v to a vertex w. The large excess, smallest label rule guarantees that before the push $e_f(v) > \Delta/2$ and either $e_f(w) \leq \Delta/2$ or $w = t$. Thus the push moves at least $\Delta/2$ units of flow, and hence decreases Φ by at least $1/2$. The initial and final values of Φ are zero, so the total number of nonsaturating pushes is at most twice the sum of the increases in Φ over the course of the algorithm. Increasing the label of a vertex by k can increase Φ by at most k. Thus relabelings account for a total increase in Φ of at most $2n^2$. A change in phase also increases Φ, by a factor of two, or at most n^2. Hence the sum of increases in Φ is at most $2n^2 + n^2(\log_2 U + 1)$, and therefore the number of nonsaturating pushes is at most $4n^2 + 2n^2(\log_2 U + 1)$. ∎

The large excess, smallest label rule can be implemented by storing the active vertices with excess exceeding $\Delta/2$ in an array of sets, as was previously described for the largest-label rule. In the former case, the index b indicates the nonempty set of smallest index. Since b decreases only when a push occurs, and then by at most 1, the total time spent updating b is $O(nm + n^2 \log U)$. From Lemmas 2.3.2 and 2.4.1 we obtain the following result:

Theorem 2.4.2 [3] The excess-scaling algorithm runs in $O(nm + n^2 \log U)$ time.

A more complicated version of excess scaling, devised by Ahuja, Orlin, and Tarjan [4], has a running time of $O(nm + n^2\sqrt{\log U})$. This algorithm uses a hybrid vertex selection rule that combines a stack-based mechanism with the "wave" approach of Tarjan [95].

make-tree(v):	Make vertex v into a one-vertex dynamic tree. Vertex v must be in no other tree.
find-root(v):	Find and return the root of the tree containing vertex v.
find-value(v):	Find and return the value of the tree arc connecting v to its parent. If v is a tree root, return infinity.
find-min(v):	Find and return the ancestor w of v such that the tree arc connecting w to its parent has minimum value along the path from v to *find-root*(v). In case of a tie, choose the vertex w closest to the tree root. If v is a tree root, return v.
change-value(v, x):	Add real number x to the value of every arc along the path from v to *find-root*(v).
link(v, w, x):	Combine the trees containing v and w by making w the parent of v and giving the new tree arc joining v and w the value x. 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 arc from v to its parent. This operation does nothing if v is a tree root.

Figure 2.5: **The dynamic tree operations.**

2.5 Use of Dynamic Trees

The two previous sections discussed ways of reducing the number of nonsaturating pushes by restricting the order of the update operations. An orthogonal approach is to reduce the *time* per nonsaturating push rather than the *number* of such pushes. The idea is to perform a succession of pushes along a single path in one operation, using a sophisticated data structure to make this possible. Observe that immediately after a nonsaturating push along an arc (v, w), (v, w) is still admissible, and we know its residual capacity. The *dynamic tree* data structure of Sleator and Tarjan [92, 93] provides an efficient way to maintain information about such arcs. We shall describe a dynamic tree version of the generic algorithm that has a running time of $O(nm \log n)$.

The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each arc of which has an associated real value. We regard a tree arc as directed toward the root, *i.e.*, from child to parent. We denote the parent of a vertex v by *parent*(v), and adopt the convention that every vertex is both an ancestor and a descendant of itself. The data structure supports the seven operations described in Figure 2.5. A sequence of l tree operations on trees of maximum size (number of vertices) k takes $O(l \log k)$ time.

In the dynamic tree algorithm, the arcs of the dynamic trees are a subset of the admissible arcs and every active vertex is a tree root. The value of an arc (v, w) is its residual capacity. Initially each vertex is made into a one-vertex dynamic tree using a *make-tree* operation.

The heart of the dynamic tree implementation is the *tree-push* operation described in Figure 2.6. This operation is applicable to an admissible arc (v, w) such that v is active. The operation adds (v, w) to the forest of dynamic trees, pushes as much flow as possible along the tree from v to the root of the tree containing w, and deletes from the forest each arc that is saturated by this flow

```
Tree-push (v, w)
Applicability: v is active and (v, w) is admissible.
Action:        link (v, w, u_f(v, w));
               parent (v) ← w;
               δ ← min {e_f (v), find-value(find-min (v));
               change-value (v, −δ);
               while v ≠ find-root(v) and find-value(find-min(v)) = 0 do begin
                     z ← find-min(v);
                     cut (z);
                     f(z, parent(z)) ← u(z, parent(z));
                     f(parent(z), z) ← −u(z, parent(z));
               end.
```

Figure 2.6: The *tree-push* operation.

change.

The dynamic tree algorithm is just the generic algorithm with the *push* operation replaced by the *tree-push* operation, with the initialization modified to make each vertex into a dynamic tree, and with a postprocessing step that extracts the correct flow value for each arc remaining in a dynamic tree. (This postprocessing takes one *find-value* operation per vertex.)

It is straightforward to show that the dynamic tree implementation is correct, by observing that it maintains the invariant that between tree-pushing operations every active vertex is a dynamic tree root and every dynamic tree arc is admissible. The following lemma bounds the number of *tree-push* operations:

Lemma 2.5.1 The number of the *tree-push* operations done by the dynamic tree implementation is $O(nm)$.

Proof: Each *tree-push* operation either saturates an arc (thus doing a saturating push) or decreases the number of active vertices by one. The number of active vertices can increase, by at most one, as a result of a *tree-push* operation that saturates an arc. By Lemma 2.2.5 there are at most nm saturating pushes. An upper bound of $2nm + n$ on the number of *tree-push* operations follows, since initially there are at most n active vertices. ∎

If we implement the dynamic tree algorithm using the *discharge* operation of Section 2.3 with *push* replaced by *tree-push*, we obtain the following result (assuming active vertex selection takes $O(1)$ time):

Theorem 2.5.2 The *discharge*-based implementation of the dynamic tree algorithm runs in $O(nm \log n)$ time.

Proof: Each *tree-pushing* operation does $O(1)$ dynamic tree operations plus $O(1)$ per arc saturated.

The theorem follows from Lemma 2.2.5, Lemma 2.3.2, and Lemma 2.5.1, since the maximum size of any dynamic tree is $O(n)$. ∎

The dynamic tree data structure can be used in combination with the FIFO, largest-label, or excess-scaling method. The resulting time bounds are $O(nm \log(n^2/m))$ for the FIFO and largest-label methods [44] and $O(nm \log(\frac{n}{m}\sqrt{\log U} + 2))$ for the fastest version of the excess-scaling method [4]. In each case, the dynamic tree method must be modified so that the trees are not too large, and the analysis needed to obtain the claimed bound is rather complicated.

Chapter 3

The Minimum-Cost Circulation Problem: Cost-Scaling

3.1 Introduction

Most polynomial-time algorithms for the minimum-cost circulation problem use the idea of scaling. This idea was introduced by Edmonds and Karp [21], who used it to develop the first polynomial-time algorithm for the problem. Scaling algorithms work by obtaining a sequence of feasible or almost-feasible solutions that are closer and closer to the optimum. The Edmonds-Karp algorithm scales capacities. Röck [88] was the first to propose an algorithm that scales costs. Later, Bland and Jensen [13] proposed a somewhat different cost-scaling algorithm, which is closer to the generalized cost scaling method discussed in this chapter.

The *cost-scaling* approach works by solving a sequence of problems, $P_0, P_1, \ldots, P_k$, on the network with original capacities but approximate costs. The cost function c_i for P_i is obtained by taking the i most significant bits of the original cost function c. The first problem P_0 has zero costs, and therefore the zero circulation is optimal. An optimal solution to problem P_{i-1} can be used to obtain an optimal solution to problem P_i in at most n maximum flow computations [13, 88]. Note that for $k = \lceil \log_2 C \rceil$, $c_k = c$. Thus the algorithm terminates in $O(n \log C)$ maximum flow computations.

Goldberg and Tarjan [39, 46, 48] proposed a *generalized cost-scaling* approach. The idea of this method (which is described in detail below) is to view the maximum amount of violation of the complementary slackness conditions as an error parameter, and to improve this parameter by a factor of two at each iteration. Initially the error is at most C, and if the error is less than $1/n$, then the current solution is optimal. Thus the generalized cost-scaling method terminates in $O(\log(nC))$ iterations. The computation done at each iteration is similar to a maximum flow computation. The traditional cost-scaling method of Röck also improves the error from iteration to iteration, but it does so indirectly, by increasing the precision of the current costs and solving the

resulting problem exactly. Keeping the original costs, as does the generalized cost-scaling approach, makes it possible to reduce the number of iterations required and to obtain strongly-polynomial running time bounds. Chapter 4 discusses a strongly-polynomial version of the generalized cost-scaling method. For further discussion of generalized versus traditional cost-scaling, see [47].

Time bounds for the cost-scaling algorithms mentioned above are as follows. The algorithms of Röck [88] and Bland and Jensen [13] run in $O(n \log(C) M(n, m, U))$ time, where $M(n, m, U)$ is the time required to compute a maximum flow on a network with n vertices, m arcs, and maximum arc capacity U. As we have seen in Chapter 2, $M = O(nm \log \min\{n^2/m, \frac{n}{m}\sqrt{\log U} + 2\})$. The fastest known implementation of the generalized cost-scaling method runs in $O(nm \log(n^2/m) \log(nC))$ time [46]. It is possible to combine cost scaling with capacity scaling. The first algorithm that combines the two scaling techniques is due to Gabow and Tarjan [33]. A different algorithm was proposed by Ahuja et al. [1]. The latter algorithm runs in $O(nm \log\log U \log(nC))$ time, which makes it the fastest known algorithm for the problem under the similarity assumption.

3.2 Approximate Optimality

A key notion is that of *approximate optimality*, obtained by relaxing the complementary slackness constraints in Theorem 1.3.3. For a constant $\epsilon \geq 0$, a pseudoflow f is said to be *ϵ-optimal with respect to a price function p* if, for every arc (v, w), we have

$$f(v, w) < u(v, w) \Rightarrow c_p(v, w) \geq -\epsilon \quad (\epsilon\text{-optimality constraint}). \tag{3.1}$$

A pseudoflow f is *ϵ-optimal* if f is ϵ-optimal with respect to some price function p.

An important property of ϵ-optimality is that if the arc costs are integers and ϵ is small enough, any ϵ-optimal circulation is minimum-cost. The following theorem, of Bertsekas [11] captures this fact.

Theorem 3.2.1 [11] **If all costs are integers and $\epsilon < 1/n$, then an ϵ-optimal circulation f is minimum-cost.**

The ϵ-optimality constraints were first published by Tardos [94] in a paper describing the first strongly polynomial algorithm for the minimum-cost circulation problem. Bertsekas [11] proposed a pseudopolynomial algorithm based upon Theorem 3.2.1; his algorithm makes use of a fixed $\epsilon < 1/n$. Goldberg and Tarjan [39, 47, 48] devised a successive approximation scheme that produces a sequence of circulations that are ϵ-optimal for smaller and smaller values of ϵ; when ϵ is small enough, the scheme terminates with an optimal circulation. We discuss this scheme below.

```
procedure min-cost(V, E, u, c);
    [initialization]
    ε ← C;
    ∀v,  p(v) ← 0;
    ∀(v, w) ∈ E,  f(v, w) ← 0;
    [loop]
    while ε ≥ 1/n do
        (ε, f, p) ← refine(ε, f, p);
    return(f);
end.
```

Figure 3.1: The generalized cost-scaling method.

3.3 The Generalized Cost-Scaling Framework

Throughout the rest of this chapter, we assume that all arc costs are integral. We give here a high-level description of the generalized cost-scaling method (see Figure 3.1). The algorithm maintains an error parameter ϵ, a circulation f and a price function p, such that f is ϵ-optimal with respect to p. The algorithm starts with $\epsilon = C$ (or alternatively $\epsilon = 2^{\lceil \log_2 C \rceil}$), with $p(v) = 0$ for all $v \in V$, and with the zero circulation. Any circulation is C-optimal. The main loop of the algorithm repeatedly reduces the error parameter ϵ. When $\epsilon < 1/n$, the current circulation is minimum-cost, and the algorithm terminates.

The task of the subroutine *refine* is to reduce the error in the optimality of the current circulation. The input to *refine* is an error parameter ϵ, a circulation f, and a price function p such that f is ϵ-optimal with respect to p. The output from *refine* is a reduced error parameter ϵ, a new circulation f, and a new price function p such that f is ϵ-optimal with respect to p. The implementations of *refine* described in this survey reduce the error parameter ϵ by a factor of two.

The correctness of the algorithm is immediate from Theorem 3.2.1, assuming that *refine* is correct. The number of iterations of *refine* is $O(\log(nC))$. This gives us the following theorem:

Theorem 3.3.1 [48] A minimum-cost circulation can be computed in the time required for $O(\log(nC))$ iterations of refine, if refine reduces ϵ by a factor of at least two.

3.4 A Generic Refinement Algorithm

In this section we describe an implementation of *refine* that is a common generalization of the generic maximum flow algorithm of Chapter 2.2 and the auction algorithm for the assignment problem [9] (first published in [10]). We call this the *generic implementation.* This implementation, proposed by Goldberg and Tarjan [48], is essentially the same as the main loop of the minimum-cost circulation algorithm of Bertsekas [11], which is also a common generalization of the maximum

```
procedure refine(ε, f, p);
    [initialization]
    ε ← ε/2;
    ∀(v, w) ∈ E do if c_p(v, w) < 0 then begin f(v, w) ← u(v, w); f(w, v) ← −u(v, w); end;
    [loop]
    while ∃ an update operation that applies do
        select such an operation and apply it;
    return(ε, f, p);
end.
```

Figure 3.2: The generic *refine* subroutine.

push(v, w).
Applicability: v is active **and** (v, w) is admissible.
Action: send $\delta = \min(e_f(v), u_f(v, w))$ units of flow from v to w.

relabel(v).
Applicability: v is active **and** $\forall w \in V$ $(u_f(v, w) = 0$ **or** $c_p(v, w) \geq 0)$.
Action: replace $p(v)$ by $\max_{(v,w)\in E_f}\{p(w) - c(v, w) - \epsilon\}$.

Figure 3.3: The update operations for the generic refinement algorithm. Compare with Figure 2.2.

flow and assignment algorithms. The ideas behind the auction algorithm can be used to give an alternative interpretation to the results of [39, 48] in terms of relaxation methods; see [12].

As we have mentioned in Section 3.3, the effect of *refine* is to reduce ϵ by a factor of two while maintaining the ϵ-optimality of the current flow f with respect to the current price function p. The generic *refine* subroutine is described on Figure 3.2. It begins by halving ϵ and saturating every arc with negative reduced cost. This converts the circulation f into an ϵ-optimal pseudoflow (indeed, into a 0-optimal pseudoflow). Then the subroutine converts the ϵ-optimal pseudoflow into an ϵ-optimal circulation by applying a sequence of the *update operations push* and *relabel*, each of which preserves ϵ-optimality.

The inner loop of the generic algorithm consists of repeatedly applying the two update operations, described in Figure 3.3, in any order, until no such operation applies. To define these operations, we need to redefine admissible arcs in the context of the minimum-cost circulation problem. Given a pseudoflow f and a price function p, we say that an arc (v, w) is *admissible* if (v, w) is a residual arc with negative reduced cost.

A *push* operation applies to an admissible arc (v, w) such that vertex v is active. It consists of pushing $\delta = \min\{e_f(v), u_f(v, w)\}$ units of flow from v to w, thereby decreasing $e_f(v)$ and $f(w, v)$ by δ and increasing $e_f(w)$ and $f(v, w)$ by δ. The push is *saturating* if $u_f(v, w) = 0$ after the push and *nonsaturating* otherwise.

A *relabel* operation applies to an active vertex v that has no exiting admissible arcs. It consists of decreasing $p(v)$ to the smallest value allowed by the ϵ-optimality constraints, namely $\max_{(v,w)\in E_f}\{-c(v,w)+p(w)-\epsilon\}$.

If an ϵ-optimal pseudoflow f is not a circulation, then either a pushing or a relabeling operation is applicable. It is easy to show that any pushing operation preserves ϵ-optimality. The next lemma gives two important properties of the relabeling operation.

Lemma 3.4.1 Suppose f is an ϵ-optimal pseudoflow with respect to a price function p and a vertex v is relabeled. Then the price of v decreases by at least ϵ and the pseudoflow f is ϵ-optimal with respect to the new price function p'.

Proof: Before the relabeling, $c_p(v,w) \geq 0$ for all $(v,w) \in E_f$, *i.e.*, $p(v) \geq p(w) - c_p(v,w)$ for all $(v,w) \in E_f$. Thus $p'(v) = \max_{(v,w)\in E_f} \{p(w) - c(v,w) - \epsilon\} \leq p(v) - \epsilon$.

To verify ϵ-optimality, observe that the only residual arcs whose reduced costs are affected by the relabeling are those of the form (v,w) or (w,v). Any arc of the form (w,v) has its reduced cost increased by the relabeling, preserving its ϵ-optimality constraint. Consider a residual arc (v,w). By the definition of p', $p'(v) \geq p(w) - c(v,w) - \epsilon$. Thus $c_{p'}(v,w) = c(v,w) + p'(v) - p(w) \geq -\epsilon$, which means that (v,w) satisfies its ϵ-optimality constraint. ∎

Since the update operations preserve ϵ-optimality, and since some update operation applies if f is not a circulation, it follows that if *refine* terminates and returns (ϵ, f, p), then f is a circulation which is ϵ-optimal with respect to p. Thus *refine* is correct.

Next we analyze the number of update operations that can take place during an execution of *refine*. We begin with a definition. The *admissible graph* is the graph $G_A = (V, E_A)$ such that E_A is the set of admissible arcs. As *refine* executes, the admissible graph changes. An important invariant is that the admissible graph remains acyclic.

Lemma 3.4.2 Immediately after a relabeling is applied to a vertex v, no admissible arcs enter v.

Proof: Let (u,v) be a residual arc. Before the relabeling, $c_p(u,v) \geq -\epsilon$ by ϵ-optimality. By Lemma 3.4.1, the relabeling decreases $p(v)$, and hence increases $c_p(u,v)$, by at least ϵ. Thus $c_p(u,v) \geq 0$ after the relabeling. ∎

Corollary 3.4.3 Throughout the running of *refine*, the admissible graph is acyclic.

Proof: Initially the admissible graph contains no arcs and is thus acyclic. Pushes obviously preserve acyclicity. Lemma 3.4.2 implies that relabelings also preserve acyclicity. ∎

Next we derive a crucial lemma, which generalizes Lemma 2.2.1.

Lemma 3.4.4 Let f be a pseudoflow and f' a circulation. For any vertex v with $e_f(v) > 0$, there is a vertex w with $e_f(w) < 0$ and a sequence of distinct vertices $v = v_0, v_1, \ldots, v_{l-1}, v_l = w$ such that $(v_i, v_{i+1}) \in E_f$ and $(v_{i+1}, v_i) \in E_{f'}$ for $0 \leq i < l$.

Proof: Let v be a vertex with $e_f(v) > 0$. Define $G_+ = (V, E_+)$, where $E_+ = \{(x,y) | f'(x,y) > f(x,y)\}$, and define $G_- = (V, E_-)$, where $E_- = \{(x,y) | f(x,y) > f'(x,y)\}$. Then $E_+ \subseteq E_f$, since $(x,y) \in E_+$ implies $f(x,y) < f'(x,y) \leq u(x,y)$. Similarly $E_- \subseteq E_{f'}$. Furthermore $(x,y) \in E_+$ if and only if $(y,x) \in E_-$ by antisymmetry. Thus to prove the lemma it suffices to show the existence in G_+ of a simple path $v = v_0, v_1 \ldots, v_l$ with $e_f(v_l) < 0$.

Let S be the set of all vertices reachable from v in G_+ and let $\overline{S} = V - S$. (Set $\overline{S}$ may be empty.) For every vertex pair $(x,y) \in S \times \overline{S}$, $f(x,y) \geq f'(x,y)$, for otherwise $y \in S$. We have

$$\begin{aligned}
0 &= \textstyle\sum_{(x,y)\in(S\times\overline{S})\cap E} f'(x,y) && \text{since } f' \text{ is a circulation} \\
&\leq \textstyle\sum_{(x,y)\in(S\times\overline{S})\cap E} f(x,y) && \text{holds term-by-term} \\
&= \textstyle\sum_{(x,y)\in(S\times\overline{S})\cap E} f(x,y) + \sum_{(x,y)\in(S\times S)\cap E} f(x,y) && \text{by antisymmetry} \\
&= \textstyle\sum_{(x,y)\in(S\times V)\cap E} f(x,y) && \text{by definition of } \overline{S} \\
&= -\textstyle\sum_{x\in S} e_f(x) && \text{by antisymmetry.}
\end{aligned}$$

But $v \in S$. Since $e_f(v) > 0$, some vertex $w \in S$ must have $e_f(w) < 0$. ∎

Using Lemma 3.4.4 we can bound the amount by which a vertex price can decrease during an invocation of *refine*.

Lemma 3.4.5 The price of any vertex v decreases by at most $3n\epsilon$ during an execution of refine.

Proof: Let $f_{2\epsilon}$ and $p_{2\epsilon}$ be the circulation and price functions on entry to *refine*. Suppose a relabeling causes the price of a vertex v to decrease. Let f be the pseudoflow and p the price function just after the relabeling. Then $e_f(v) > 0$. Let $v = v_0, v_1, \ldots, v_l = w$ with $e_f(w) < 0$ be the vertex sequence satisfying Lemma 3.4.4 for f and $f' = f_{2\epsilon}$.

The ϵ-optimality of f implies

$$-l\epsilon \leq \sum_{i=0}^{l-1} c_p(v_i, v_{i+1}) = p(v) - p(w) + \sum_{i=0}^{l-1} c(v_i, v_{i+1}). \tag{3.2}$$

The 2ϵ-optimality of $f_{2\epsilon}$ implies

$$-2l\epsilon \leq \sum_{i=0}^{l-1} c_{p_{2\epsilon}}(v_{i+1}, v_i) = p_{2\epsilon}(w) - p_{2\epsilon}(v) + \sum_{i=0}^{l-1} c(v_{i+1}, v_i). \tag{3.3}$$

But $\sum_{i=0}^{l-1} c(v_i, v_{i+1}) = -\sum_{i=0}^{l-1} c(v_{i+1}, v_i)$ by cost antisymmetry. Furthermore, $p(w) = p_{2\epsilon}(w)$ since during *refine*, the initialization step is the only one that makes the excess of some vertices negative, and a vertex with negative excess has the same price as long as its excess remains negative. Adding inequalities (3.2) and (3.3) and rearranging terms thus gives

$$p(v) \geq p_{2\epsilon}(v) - 3l\epsilon > p_{2\epsilon}(v) - 3n\epsilon. \quad \blacksquare$$

Now we count update operations. The following lemmas are analogous to Lemmas 2.2.4, 2.2.5 and 2.2.6.

Lemma 3.4.6 The number of relabelings during an execution of refine is at most $3n$ per vertex and $3n(n-1)$ in total.

Lemma 3.4.7 The number of saturating pushes during an execution of refine is at most $3nm$.

Proof: For an arc (v, w), consider the saturating pushes along this arc. Before the first such push can occur, vertex v must be relabeled. After such a push occurs, v must be relabeled before another such push can occur. But v is relabeled at most $3n$ times. Summing over all arcs gives the desired bound. $\blacksquare$

Lemma 3.4.8 The number of nonsaturating pushes during one execution of refine is at most $3n^2(m+n)$.

Proof: For each vertex v, let $\Phi(v)$ be the number of vertices reachable from v in the current admissible graph G_A. Let $\Phi = 0$ if there are no active vertices, and let $\Phi = \sum\{\Phi(v) | v \text{ is active}\}$ otherwise. Throughout the running of *refine*, $\Phi \geq 0$. Initially $\Phi \leq n$, since G_A has no arcs.

Consider the effect on Φ of update operations. A nonsaturating push decreases Φ by at least one, since G_A is always acyclic by Corollary 3.4.3. A saturating push can increase Φ by at most n, since at most one inactive vertex becomes active. If a vertex v is relabeled, Φ also can increase by at most n, since $\Phi(w)$ for $w \neq v$ can only decrease by Lemma 3.4.2. The total number of nonsaturating pushes is thus bounded by the initial value of Φ plus the total increase in Φ throughout the algorithm, *i.e.*, by $n + 3n^2(n-1) + 3n^2m \leq 3n^2(m+n)$. $\blacksquare$

3.5 Efficient Implementation

As in the case of the generic maximum flow algorithm, we can obtain an especially efficient version of refine by choosing the order of the update operations carefully.

Local ordering of the basic operations is achieved using the data structures and the *discharge* operation of Section 2.3. The *discharge* operation is the same as the one in Figure 2.3, but uses the

```
procedure first-active;
    let L be a list of all vertices;
    let v be the first vertex in L;
    while ∃ an active vertex do begin
        if v is active then begin
            discharge(v);
            if the discharge has relabeled v then
              move v to the front of L;
        end;
        else replace v by the vertex after v on L,
    end;
end.
```

Figure 3.4: The *first-active* method.

minimum-cost circulation versions of the update operations (see Figure 3.3). As in the maximum flow case, it is easy to show that *discharge* applies the relabeling operation correctly. The overall running time of *refine* is $O(nm)$ plus $O(1)$ per nonsaturating push plus the time needed for active vertex selection.

In the maximum flow case, one of the vertex selection methods we considered was the largest-label method. In the minimum-cost circulation case, a good method is *first-active* [47], which is a generalization of the largest-label method. The idea of the method is to process vertices in topological order with respect to the admissible graph.

The first-active method maintains a list L of all the vertices of G, in topological order with respect to the current admissible graph G_A, *i.e.*, if (v, w) is an arc of G_A, v appears before w in L. Initially L contains the vertices of G in any order. The method consists of repeating the following step until there are no active vertices: Find the first active vertex on L, say v, apply a *discharge* operation to v, and move v to the front of L if the *discharge* operation has relabeled v.

In order to implement this method, we maintain a *current vertex* v of L, which is the next candidate for discharging. Initially v is the first vertex on L. The implementation, described in Figure 3.4, repeats the following step until there are no active vertices: If v is active, apply a discharge operation to it, and if this operation relabels v, move v to the front of L; otherwise (*i.e.*, v is inactive), replace v by the vertex currently after it on L. Because the reordering of L maintains a topological ordering with respect to G_A, no active vertex precedes v on L. This implies that the implementation is correct.

Define a *phase* as a period of time that begins with v equal to the first vertex on L and ends when the next relabeling is performed (or when the algorithm terminates).

Lemma 3.5.1 The first-active method terminates after $O(n^2)$ phases.

Proof: Each phase except the last ends with a relabeling operation. ∎

Theorem 3.5.2 [48] The first-active implementation of refine runs in $O(n^3)$ time, giving an $O(n^3 \log(nC))$ bound for finding a minimum-cost circulation.

Proof: Lemma 3.5.1 implies that there are $O(n^3)$ nonsaturating pushes (one per vertex per phase) during an execution of *refine*. The time spent manipulating L is $O(n)$ per phase, for a total of $O(n^3)$. All other operations require a total of $O(nm)$ time. ∎

A closely related strategy for selecting the next vertex to process is the *wave* method [39, 47, 48], which gives the same $O(n^3)$ running time bound for *refine*. (A similar pseudopolynomial algorithm, without the use of scaling and missing some of the implementation details, was developed independently in [11].) The only difference between the first-active method and the wave method is that the latter, after moving a vertex v to the front of L, replaces v by the vertex after the *old* position of v; if v is the last vertex on L, v is replaced by the first vertex on L.

As in the maximum flow case, the dynamic tree data structure can be used to obtain faster implementations of *refine*. A dynamic tree implementation of the generic version of *refine* analogous to the maximum flow algorithm discussed in Section 2.5 runs in $O(nm \log n)$ time [48]. A dynamic tree implementation of either the first-active method or the wave method runs in $O(nm \log(n^2/m))$ time [47]. In the latter implementation, a second data structure is needed to maintain the list L. The details are somewhat involved.

3.6 Refinement Using Blocking Flows

An alternative way to implement the refine subroutine is to generalize Dinic's approach to the maximum flow problem. Goldberg and Tarjan [47, 48] showed that refinement can be carried out by solving a sequence of $O(n)$ blocking flow problems on acyclic networks (*i.e.*, on networks for which the residual graph of the zero flow is acyclic); this extends Dinic's result, which reduces a maximum flow problem to $n-1$ blocking flow problems on layered networks. In this section we describe the Goldberg-Tarjan algorithm. At the end of this section, we make a few comments about blocking flow algorithms.

To describe the blocking flow version of refine we need some standard definitions. Consider a flow network (G, u, s, t). A flow f is *blocking* if any path from s to t *in the residual graph of zero flow* contains a saturated arc, *i.e.*, an arc (v, w) such that $u_f(v, w) = 0$. A maximum flow is blocking, but not conversely. A directed graph is *layered* if its vertices can be assigned integer layers in such a way that $layer(v) = layer(w) + 1$ for every arc (v, w). A layered graph is acyclic but not conversely.

An observation that is crucial to this section is as follows. Suppose we have a pseudoflow f and a price function p such that the vertices can be partitioned into two sets, S and $\overline{S}$, such that no admissible arc leads from a vertex in S to a vertex in $\overline{S}$; in other words, for every residual arc

```
procedure refine(ε, f, p);
    [initialization]
    ε ← ε/2;
    ∀(v, w) ∈ E do if c_p(v, w) < 0 then f(v, w) ← u(v, w);
    [loop]
    while f is not a circulation do begin
        S ← {v ∈ V | ∃u ∈ V such that e_f(u) > 0 and v is reachable from u in G_A};
        ∀v ∈ S, p(v) ← p(v) − ε;
        let N be the network formed from G_A by adding a source s, a sink t,
            an arc (s, v) of capacity e_f(v) for each v ∈ V with e_f(v) > 0, and
            an arc (v, t) of capacity −e_f(v) for each v ∈ V with e_f(v) < 0;
        find a blocking flow b on N;
        ∀(v, w) ∈ E_A, f(v, w) ← f(v, w) + b(v, w);
    end;
    return(ε, f, p);
end.
```

Figure 3.5: The blocking *refine* subroutine.

$(v, w) \in E_f$ such that $v \in S$ and $w \in \overline{S}$, we have $c_p(v, w) \geq 0$. Define p' to be equal to p on $\overline{S}$ and to $p - \epsilon$ on S. It is easy to see that replacing p by p' does not create any new residual arc with reduced cost less than $-\epsilon$. The blocking flow method augments by a blocking flow to create a partitioning of vertices as described above, and modifies the price function by replacing p by p'.

Figure 3.5 describes an implementation of *refine* that reduces ϵ by a factor of two by computing $O(n)$ blocking flows. This implementation reduces ϵ by a factor of two, saturates all admissible arcs, and then modifies the resulting pseudoflow (while maintaining ϵ-optimality with respect to the current price function) until it is a circulation. To modify the pseudoflow, the method first partitions the vertices of G into two sets S and $\overline{S}$, such that S contains all vertices reachable in the current admissible graph G_A from vertices of positive excess. Vertices in S have their prices decreased by ϵ. Next, an auxiliary network N is constructed by adding to G_A a source s, a sink t, an arc (s, v) of capacity $e_f(v)$ for each vertex v with $e_f(v) > 0$, and an arc (v, t) of capacity $-e_f(v)$ for each vertex with $e_f(v) < 0$. An arc $(v, w) \in E_A$ has capacity $u_f(v, w)$ in N. A blocking flow b on N is found. Finally, the pseudoflow f is replaced by the pseudoflow $f'(v, w) = f(v, w) + b(v, w)$ for $(v, w) \in E$.

The correctness of the blocking flow method follows from the next lemma, which can be proved by induction on the number of iterations of the method.

Lemma 3.6.1 The set S computed in the inner loop contains only vertices v with $e_f(v) \geq 0$. At the beginning of an iteration of the loop, f is an ϵ-optimal pseudoflow with respect to the price function p. Decreasing the prices of vertices in S by ϵ preserves the ϵ-optimality of f. The admissible graph remains acyclic throughout the algorithm.

The bound on the number of iterations of the method follows from Lemma 3.4.5 and the fact that the prices of the vertices with deficit remain unchanged, while the prices of the vertices with excess decrease by ϵ during every iteration.

Lemma 3.6.2 The number of iterations of the inner loop in the blocking flow implementation of refine is at most $3n$.

Since the running time of an iteration of the blocking flow method is dominated by the time needed for a blocking flow computation, we have the following theorem.

Theorem 3.6.3 [48] The blocking flow implementation of refine runs in $O(nB(n,m))$ time, giving an $O(nB(n,m)\log(nC))$ bound for finding a minimum-cost circulation, where $B(n,m)$ is the time needed to find a blocking flow in an acyclic network with n vertices and m arcs.

The fastest known sequential algorithm for finding a blocking flow on an acyclic network is due to Goldberg and Tarjan [47] and runs in $O(m\log(n^2/m))$ time. Thus, by Theorem 3.6.3, we obtain an $O(nm\log(n^2/m)\log(nC))$ time bound for the minimum-cost circulation problem. This is the same as the fastest known implementation of the generic refinement method.

There is a crucial difference between Dinic's maximum flow algorithm and the blocking flow version of *refine*. Whereas the former finds blocking flows in layered networks, the latter must find blocking flows in acyclic networks, an apparently harder task. Although for sequential computation the acyclic case seems to be no harder than the layered case (the best known time bound is $O(m\log(n^2/m)$ for both), this is not true for sequential computation. The Shiloach-Vishkin PRAM blocking flow algorithm [91] for layered networks runs in $O(n^2\log n)$ time using $O(n^2)$ memory and n processors. The fastest known PRAM algorithm for acyclic networks, due to Goldberg and Tarjan [45], runs in the same $O(n^2\log n)$ time bound but uses $O(nm)$ memory and m processors.

Chapter 4

Strongly Polynomial Algorithms Based on Cost-Scaling

4.1 Introduction

The question of whether the minimum-cost circulation problem has a strongly polynomial algorithm was posed in 1972 by Edmonds and Karp [21] and resolved only in 1984 by Tardos [94]. Her result led to the discovery of a number of strongly polynomial algorithms for the problem [29, 36, 79]. In this chapter we discuss several strongly polynomial algorithms based on cost scaling; in the next, we explore capacity-scaling algorithms, including strongly polynomial ones. Of the known strongly polynomial algorithms, the asymptotically fastest is the capacity-scaling algorithm of Orlin [79].

We begin by describing a modification of the generalized cost-scaling method that makes it strongly polynomial [47]. Then we describe the minimum-mean cycle-canceling algorithm of Goldberg and Tarjan [46]. This simple algorithm is a specialization of Klein's cycle-canceling algorithm [69]; it does not use scaling, but its analysis relies on ideas related to cost scaling.

4.2 Fitting Price Functions and Tight Error Parameters

In order to obtain strongly polynomial bounds on the generalized cost-scaling method, we need to take a closer look at the notion of ϵ-optimality defined in Section 3.2. The definition of ϵ-optimality motivates the following two problems:

1. Given a pseudoflow f and a constant $\epsilon \geq 0$, find a price function p such that f is ϵ-optimal with respect to p, or show that there is no such price function (*i.e.*, that f is not ϵ-optimal).

2. Given a pseudoflow f, find the the smallest $\epsilon \geq 0$ such that f is ϵ-optimal. For this ϵ, we say that f is *ϵ-tight.*

The problem of finding an optimal price function given an optimal circulation is the special case of Problem 1 with $\epsilon = 0$. We shall see that the first problem can be reduced to a shortest path problem, and that the second problem requires the computation of a cycle of minimum average arc cost.

To address these problems, we need some results about shortest paths and shortest path trees (see e.g. [97]). Let G be a directed graph with a distinguished *source vertex* s from which every vertex is reachable and a *cost* $c(v, w)$ on every arc (v, w). For a spanning tree T rooted at s, the *tree cost function* $d : V \to R$ is defined recursively as follows: $d(s) = 0$, $d(v) = d(parent(v)) + c(parent(v), v)$ for $v \in V - \{s\}$, where $parent(v)$ is the parent of v in T. A spanning tree T rooted at s is a *shortest path tree* if and only if, for every vertex v, the path from s to v in T is a minimum-cost path from s to v in G, *i.e.*, $d(v)$ is the cost of a minimum-cost path from s to v.

Lemma 4.2.1 (see e.g. [97]) Graph G contains a shortest path tree if and only if G does not contain a negative-cost cycle. A spanning tree T rooted at s is a shortest path tree if and only if $c(v, w) + d(v) \geq d(w)$ for every arc (v, w) in G.

Consider Problem 1: given a pseudoflow f and a nonnegative ϵ, find a price function p with respect to which f is ϵ-optimal, or show that f is not ϵ-optimal. Define a new cost function $c^{(\epsilon)} : E \to R$ by $c^{(\epsilon)}(v, w) = c(v, w) + \epsilon$. Extend the residual graph G_f by adding a single vertex s and arcs from it to all other vertices to form an *auxiliary graph* $G_{aux} = (V_{aux}, E_{aux}) = (V \cup \{s\}, E_f \cup (\{s\} \times V))$. Extend $c^{(\epsilon)}$ to G_{aux} by defining $c^{(\epsilon)}(s, v) = 0$ for every arc (s, v), where $v \in V$. Note that every vertex is reachable from s in G_{aux}.

Theorem 4.2.2 Pseudoflow f is ϵ-optimal if and only if G_{aux} (or equivalently G_f) contains no cycle of negative $c^{(\epsilon)}$-cost. If T is any shortest path tree of G_{aux} (rooted at s) with respect to the arc cost function $c^{(\epsilon)}$, and d is the associated tree cost function, then f is ϵ-optimal with respect to the price function p defined by $p(v) = d(v)$ for all $v \in V$.

Proof: Suppose f is ϵ-optimal. Any cycle in G_{aux} is a cycle in G_f, since vertex s has no incoming arcs. Let Γ be a cycle of length l in G_{aux}. Then $c(\Gamma) \geq -l\epsilon$, which implies $c^{(\epsilon)}(\Gamma) = c(\Gamma) + l\epsilon \geq 0$. Therefore G_{aux} contains no cycle of negative $c^{(\epsilon)}$-cost.

Suppose G_{aux} contains no cycle of negative $c^{(\epsilon)}$-cost. Then, by Lemma 4.2.1, G_{aux} has some shortest path tree rooted at s. Let T be any such tree and let d be the tree cost function. By Lemma 4.2.1, $c^{(\epsilon)}(v, w) + d(v) \geq d(w)$ for all $(v, w) \in E_f$, which is equivalent to $c(v, w) + d(v) - d(w) \geq -\epsilon$ for all $(v, w) \in E_f$. But these are the ϵ-optimality constraints for the price function $p = d$. Thus f is ϵ-optimal with respect to p. ∎

Using Theorem 4.2.2, we can solve Problem 1 by constructing the auxiliary graph G_{aux} and finding either a shortest path tree or a negative-cost cycle. Constructing G_{aux} takes $O(m)$ time. Finding a shortest path tree or a negative-cost cycle takes $O(nm)$ time using the Bellman-Ford shortest path algorithm (see e.g. [97]).

Let us turn to Problem 2: given a pseudoflow f, find the ϵ such that f is ϵ-tight. We need a definition. For a directed graph G with arc cost function c, the *minimum cycle mean* of G, denoted by $\mu(G, c)$, is the minimum, over all cycles Γ in G, of the mean cost of Γ, defined to be the total arc cost $c(\Gamma)$ of Γ divided by the number of arcs it contains. The connection between minimum cycle means and tight error parameters is given by the following theorem, which was discovered by Engel and Schneider [22] and later by Goldberg and Tarjan [48]:

Theorem 4.2.3 [22] Suppose a pseudoflow f is not optimal. Then f is ϵ-tight for $\epsilon = -\mu(G_f, c)$.

Proof: Assume f is not optimal. Consider any cycle Γ in G_f. Let the length of Γ be l. For any ϵ, let $c^{(\epsilon)}$ be the cost function defined above: $c^{(\epsilon)}(v, w) = c(v, w) + \epsilon$ for $(v, w) \in E_f$. Let ϵ be such that f is ϵ-tight, and let $\mu = \mu(G_f, c)$. By Theorem 4.2.2, $0 \leq c^{(\epsilon)}(\Gamma) = c(\Gamma) + l\epsilon$, *i.e.*, $c(\Gamma)/l \geq -\epsilon$. Since this is true for any cycle Γ, $\mu \geq -\epsilon$, *i.e.*, $\epsilon \geq -\mu$. Conversely, for any cycle Γ, $c(\Gamma)/l \geq \mu$, which implies $c^{(-\mu)}(\Gamma) \geq 0$. By Theorem 4.2.2, this implies $-\mu \geq \epsilon$. ∎

Karp [65] observed that the minimum mean cycle can be computed in $O(nm)$ time by extracting information from a single run of the Bellman-Ford shortest path algorithm. This gives the fastest known strongly polynomial algorithm for computing the minimum cycle mean. The fastest scaling algorithm is the $O(\sqrt{n}m \log(nC))$-time algorithm of Orlin and Ahuja [83]. Since we are interested here in strongly polynomial algorithms, we shall use Karp's bound of $O(nm)$ as an estimate of the time to compute a minimum cycle mean.

The following observation is helpful in the analysis to follow. Suppose f is an ϵ-tight pseudoflow and $\epsilon > 0$. Let p be a price function such that f is ϵ-optimal with respect to p, and let Γ be a cycle in G_f with mean cost $-\epsilon$. Since $-\epsilon$ is a lower bound on the reduced cost of an arc in G_f, every arc of Γ must have reduced cost exactly $-\epsilon$.

4.3 Fixed Arcs

The main observation that leads to strongly polynomial cost-scaling algorithms for the minimum-cost circulation problem is the following result of Tardos [94]: if the absolute value of the reduced cost of an arc is significantly greater then the current error parameter ϵ, then the value of any optimal circulation on this arc is the same as the value of the current circulation. The following theorem is a slight generalization of this result (to get the original result, take $\epsilon' = 0$). This theorem can be used in two ways. The first is to drop the capacity constraint for an arc of large reduced

cost. This approach is used in [94]. The second, discussed below, is to consider the arcs that have the same flow value in every ϵ-optimal circulation for the current value of the error parameter ϵ and to notice that the flow through these arcs will not change. This approach is used in [47, 46].

Theorem 4.3.1 [94] Let $\epsilon > 0$, $\epsilon' \geq 0$ be constants. Suppose that a circulation f is ϵ-optimal with respect to a price function p, and that there is an arc $(v, w) \in E$ such that $|c_p(v, w)| \geq n(\epsilon + \epsilon')$. Then, for any ϵ'-optimal circulation f', we have $f(v, w) = f'(v, w)$.

Proof: By antisymmetry, it is enough to prove the theorem for the case $c_p(v, w) \geq n(\epsilon + \epsilon')$. Let f' be a circulation such that $f'(v, w) \neq f(v, w)$. Since $c_p(v, w) > \epsilon$, the flow f through the arc (v, w) must be as small as the capacity constraints allow, namely $-u(w, v)$, and therefore $f'(v, w) \neq f(v, w)$ implies $f'(v, w) > f(v, w)$. We show that f' is not ϵ'-optimal, and the theorem follows.

Consider $G_> = \{(x, y) \in E | f'(x, y) > f(x, y)\}$. Note that $G_>$ is a subgraph of G_f, and (v, w) is an arc of $G_>$. Since f and f' are circulations, $G_>$ must contain a simple cycle Γ that passes through (v, w). Let l be the length of Γ. Since all arcs of Γ are residual arcs, the cost of Γ is at least

$$c_p(v, w) - (l - 1)\epsilon \geq n(\epsilon + \epsilon') - (n - 1)\epsilon > n\epsilon'.$$

Now consider the cycle $\overline{\Gamma}$ obtained by reversing the arcs on Γ. Note that $\overline{\Gamma}$ is a cycle in $G_< = \{(x, y) \in E | f'(x, y) < f(x, y)\}$ and is therefore a cycle in $G_{f'}$. By antisymmetry, the cost of $\overline{\Gamma}$ is less than $-n\epsilon'$ and thus the mean cost of $\overline{\Gamma}$ is less than $-\epsilon'$. But Theorem 4.2.3 implies that f' is not ϵ'-optimal. ∎

To state an important corollary of Theorem 4.3.1, we need the following definition. We say that an arc $(v, w) \in E$ is *ϵ-fixed* if the flow through this arc is the same for all ϵ-optimal circulations.

Corollary 4.3.2 [47] Let $\epsilon > 0$, suppose f is ϵ-optimal with respect to a price function p, and suppose that (v, w) is an arc such that $|c_p(v, w)| \geq 2n\epsilon$. Then (v, w) is ϵ-fixed.

Define F_ϵ to be the set of ϵ-fixed arcs. Since the generalized cost-scaling method decreases ϵ, an arc that becomes ϵ-fixed stays ϵ-fixed. We show that when ϵ decreases by a factor of $2n$, a new arc becomes ϵ-fixed.

Lemma 4.3.3 Assume $\epsilon' \leq \frac{\epsilon}{2n}$. Suppose that there exists an ϵ-tight circulation f. Then $F_{\epsilon'}$ properly contains F_ϵ.

Proof: Since every ϵ'-optimal circulation is ϵ-optimal, we have $F_\epsilon \subseteq F_{\epsilon'}$. To show that the containment is proper, we have to show that there is an ϵ'-fixed arc that is not ϵ-fixed.

Since the circulation f is ϵ-tight, there exists a price function p such that f is ϵ-optimal with respect to p, and there exists a simple cycle Γ in G_f every arc of which has reduced cost $-\epsilon$. (See Section 4.2.) Since increasing f along Γ preserves ϵ-optimality, the arcs of Γ are not ϵ-fixed.

We show that at least one arc of Γ is ϵ'-fixed. Let f' be a circulation that is ϵ'-optimal with respect to some price function p'. Since the mean cost of Γ is $-\epsilon$, there is an arc (v, w) of Γ with $c_{p'}(v, w) \leq -\epsilon \leq -2n\epsilon'$. By Corollary 4.3.2, the arc (v, w) is ϵ'-fixed. ∎

In the next section we show how to use this lemma to get a strongly polynomial bound for a variation of the generalized cost-scaling method.

4.4 The Strongly Polynomial Framework

The minimum-cost circulation framework of Section 3.3 has the disadvantage that the number of iterations of *refine* depends on the magnitudes of the costs. If the costs are huge integers, the method need not run in time polynomial in n and m; if the costs are irrational, the method need not even terminate. In this section we show that a natural modification of the generalized cost-scaling approach produces strongly polynomial algorithms. The running time bounds we derived for algorithms based on the approach of Section 3.3 remain valid for the modified approach presented in this section. The main idea of this modification is to improve ϵ periodically by finding a price function that fits the current circulation better than the current price function. This idea can also be used to improve the practical performance of the method.

The changes needed to make the generalized cost-scaling approach strongly polynomial, suggested by Lemma 4.3.3, are to add an extra computation to the main loop of the algorithm and to change the termination condition. Before calling *refine* to reduce the error parameter ϵ, the new method computes the value λ and a price function p_λ such that the current circulation f is λ-tight with respect to p_λ. The strongly polynomial method is described on Figure 4.1. The value of λ and the price function p_λ in line $(*)$ are computed as described in Section 4.2. The algorithm terminates when the circulation f is optimal, *i.e.*, $\lambda = 0$.

The time to perform line $(*)$ is $O(nm)$. (See Section 4.2.) Since all the implementations of *refine* that we have considered have a time bound greater than $O(nm)$, the time per iteration in the new version of the algorithm exceeds the time per iteration in the original version by less than a constant factor. Since each iteration at least halves ϵ, the bound of $O(\log(nC))$ on the number of iterations derived in Chapter 3 remains valid, assuming that the costs are integral. For arbitrary real-valued costs, we shall derive a bound of $O(m \log n)$ on the number of iterations.

Theorem 4.4.1 [46] The total number of iterations of the while loop in procedure *min-cost* is $O(m \log n)$.

Proof: Consider a time during the execution of the algorithm. During the next $O(\log n)$ iterations,

```
procedure min-cost(V, E, u, c);
    [initialization]
    ε ← C;
    ∀v,  p(v) ← 0;
    ∀(v, w) ∈ E,  f(v, w) ← 0;
    [loop]
    while ε > 0 do begin
(*)     find λ and p_λ such that f is λ-tight with respect to p_λ;
        if λ > 0 then (ε, f, p) ← refine(λ, f, p_λ)
        else return(f);
end.
```

Figure 4.1: The strongly polynomial algorithm.

either the algorithm terminates, or the error parameter is reduced by a factor of $2n$. In the latter case, Lemma 4.3.3 implies that an arc becomes fixed. If all arcs become fixed, the algorithm terminates in one iteration of the loop. Therefore the total number of iterations is $O(m \log n)$. ∎

The best strongly polynomial implementation of the generalized cost-scaling method [47], based on the dynamic tree implementation of *refine*, runs in $O(nm^2 \log(n^2/m) \log n)$ time.

4.5 Cycle-Canceling Algorithms

The ideas discussed in Sections 4.2 – 4.4 are quite powerful. In this section we use these ideas to show that a simple cycle-canceling algorithm of Klein [69] becomes strongly polynomial if a careful choice is made among possible cycles to cancel. Klein's algorithm consists of repeatedly finding a residual cycle of negative cost and sending as much flow as possible around the cycle. This algorithm can run for an exponential number of iterations if the capacities and costs are integers, and it need not terminate if the capacities are irrational [25]. Goldberg and Tarjan [46] showed that if a cycle with the minimum mean cost is canceled at each iteration, the algorithm becomes strongly polynomial. We call the resulting algorithm the *minimum-mean cycle-canceling* algorithm.

The minimum-mean cycle-canceling algorithm is closely related to the shortest augmenting path maximum flow algorithm of Edmonds and Karp [21]. The relationship is as follows. If a maximum flow problem is formulated as a minimum-cost circulation problem in a standard way, then Klein's cycle-canceling algorithm corresponds exactly to the Ford-Fulkerson maximum flow algorithm, and the minimum-mean cycle-canceling algorithm corresponds exactly to the Edmonds-Karp algorithm. The minimum-mean cycle-canceling algorithm can also be interpreted as a steepest descent method using the L_1 metric.

For a circulation f we define $\epsilon(f)$ to be zero if f is optimal and to be the unique number $\epsilon' > 0$ such that f is ϵ'-tight otherwise. We use $\epsilon(f)$ as a measure of the quality of f. Let f be an arbitrary

circulation, let $\epsilon = \epsilon(f)$, and let p be a price function with respect to which f is ϵ-optimal. Holding ϵ and p fixed, we study the effect on $\epsilon(f)$ of a minimum-mean cycle cancellation that modifies f. Since all arcs on a minimum-mean cycle have negative reduced cost with respect to p, cancellation of such a cycle does not introduce a new residual arc with negative reduced cost, and hence $\epsilon(f)$ does not increase.

Lemma 4.5.1 A sequence of m minimum-mean cycle cancellations reduces $\epsilon(f)$ to at most $(1-1/n)\epsilon$, *i.e.*, to at most $1-1/n$ times its original value.

Proof: Let p a price function such that f is ϵ-tight with respect to p. Holding ϵ and p fixed, we study the effect on the admissible graph G_A (with respect to the circulation f and price function p) of a sequence of m minimum-mean cycle cancellations that modify f. Initially every arc $(v, w) \in E_A$ satisfies $c_p(v, w) \geq -\epsilon$. Canceling a cycle all of whose arcs are in E_A adds only arcs of positive reduced cost to E_f and deletes at least one arc from E_A. We consider two cases.

Case 1: None of the cycles canceled contains an arc of nonnegative reduced cost. Then each cancellation reduces the size of E_A, and after m cancellations E_A is empty, which implies that f is optimal, *i.e.*, $\epsilon(f) = 0$. Thus the lemma is true in this case.

Case 2: Some cycle canceled contains an arc of nonnegative reduced cost. Let Γ be the first such cycle canceled. Every arc of Γ has a reduced cost of at least $-\epsilon$, one arc of Γ has a nonnegative reduced cost, and the number of arcs in Γ is at most n. Therefore the mean cost of Γ is at least $-(1-1/n)\epsilon$. Thus, just before the cancellation of Γ, $\epsilon(f) \leq (1-1/n)\epsilon$ by Theorem 4.2.3. Since $\epsilon(f)$ never increases, the lemma is true in this case as well. ∎

Lemma 4.5.1 is enough to derive a polynomial bound on the number of iterations, assuming that all arc costs are integers.

Theorem 4.5.2 [46] If all arc costs are integers, then the minimum-mean cycle-canceling algorithm terminates after $O(nm\log(nC))$ iterations.

Proof: The lemma follows from Lemmas 3.2.1 and 4.5.1 and the observation that the initial circulation is C-optimal. ∎

To obtain a strongly polynomial bound, we use the ideas of Section 4.3. The proof of the next theorem uses the following inequality:

$(1-\frac{1}{n})^{n(\ln n+1)} \leq \frac{1}{2n}$ for $n \geq 2$.

Theorem 4.5.3 [46] For arbitrary real-valued arc costs, the minimum-mean cycle-canceling algorithm terminates after $O(nm^2\log n)$ cycle cancellations.

Proof: Let $k = m(n\lceil \ln n + 1\rceil)$. Divide the iterations into groups of k consecutive iterations. We claim that each group of iterations fixes the flow on a distinct arc (v, w), *i.e.*, iterations after those in the group do not change $f(v, w)$. The theorem is immediate from the claim.

To prove the claim, consider any group of iterations. Let f be the flow before the first iteration of the group, f' the flow after the last iteration of the group, $\epsilon = \epsilon(f)$, $\epsilon' = \epsilon(f')$, and let p' be a price function for which f' satisfies the ϵ'-optimality constraints. Let Γ be the cycle canceled in the first iteration of the group. By Lemma 4.5.1, the choice of k implies that $\epsilon' \leq \epsilon(1 - \frac{1}{n})^{n\lceil \ln n+1\rceil} \leq \frac{\epsilon}{2n}$. Since the mean cost of Γ is $-\epsilon$, some arc on Γ, say (v, w), must have $c_{p'}(v, w) \leq -\epsilon \leq -2n\epsilon'$. By Corollary 4.3.2, the flow on (v, w) will not be changed by iterations after those in the group. But $f(v, w)$ *is* changed by the first iteration in the group, which cancels Γ. Thus each group fixes the flow on a distinct arc. ∎

Theorem 4.5.4 [46] **The minimum-mean cycle-canceling algorithm runs in $O(n^2m^3 \log n)$ time on networks with arbitrary real-valued arc costs, and in $O(n^2m^2 \min\{\log(nC), m \log n\})$ time on networks with integer arc costs.**

Proof: Immediate from Theorems 4.5.2 and 4.5.3. ∎

Although the minimum-mean cycle-canceling algorithm seems to be of mostly theoretical interest, it has a variant that is quite efficient. This variant maintains a price function and, instead of canceling the minimum-mean-cost residual cycle, cancels residual cycles composed entirely of negative reduced cost arcs; if no such cycle exists, the algorithm updates the price function to improve the error parameter ϵ. An implementation of the algorithm using the dynamic tree data structure runs in $O(nm \log n \log(nC))$ time. See [46] for details.

The techniques used to analyze the minimum-mean cycle-canceling algorithm also provide an approach to making the primal network simplex algorithm for the minimum-cost circulation problem more efficient. Tarjan [98] has discovered a pivot rule that gives a bound of $O(n^{\frac{1}{2}\log n + O(1)})$ on the number of pivots and the total running time. This is the first known subexponential time bound. For the extended primal network simplex method in which cost-increasing as well as cost-decreasing pivots are allowed, he has obtained a strongly polynomial time bound. In contrast, the *dual* network simplex method is known to have strongly polynomial variants, as mentioned in Chapter 5.

Barahona and Tardos [7] have exhibited another cycle-canceling algorithm that runs in polynomial time. Their algorithm is based on an algorithm of Weintraub [102], which works as follows. Consider the improvement in the cost of a circulation obtained by canceling a negative cycle. Since a symmetric difference of two circulations can be decomposed into at most m cycles, canceling the cycle that gives the best improvement reduces the difference between the current and the optimal values of the cost by a factor of $(1 - 1/m)$. If the input data is integral, only a polynomial number

of such improvements can be made until an optimal solution is obtained. Finding the cycle that gives the best improvement is NP-hard, however. Weintraub shows how to find a collection of cycles whose cancellation reduces the cost by at least as much as the best improvement achievable by canceling a single cycle. His method requires a superpolynomial number of applications of an algorithm for the assignment problem. Each such application yields a minimum-cost collection of vertex-disjoint cycles. Barahona and Tardos [7] have shown that the algorithm can be modified so that the required collection of cycles is found in at most m assignment computations. The resulting minimum-cost circulation algorithm runs in polynomial time.

Chapter 5

Capacity-Scaling Algorithms

5.1 Introduction

In this chapter, we survey minimum-cost circulation algorithms based on capacity scaling. We concentrate on two algorithms: the first polynomial-time algorithm, that of Edmonds and Karp [21], who introduced the idea of scaling, and the algorithm of Orlin [79]. The latter is an extension of the Edmonds-Karp algorithm and is the fastest known strongly polynomial algorithm.

In this chapter we shall consider the (uncapacitated) transshipment problem, which is equivalent to the minimum-cost circulation problem. This simplifies the presentation considerably. We begin by describing a generic augmenting path algorithm, which we call the *minimum-cost augmentation* algorithm, that is the basis of most capacity-scaling algorithms. The algorithm is due to Jewel [60], Busacker and Gowen [14], and Iri [57]. The use of dual variables as described below was proposed independently by Edmonds and Karp [21] and Tomizawa [99]. The idea of the algorithm is to maintain a pseudoflow f that satisfies the complementary slackness constraints while repeatedly augmenting flow to gradually get rid of all excesses. Two observations justify this method: (1) augmenting flow along a minimum-cost path preserves the invariant that the current pseudoflow has minimum cost among those pseudoflows with the same excess function; (2) a shortest path computation suffices both to find a path along which to move flow and to find price changes to preserve complementary slackness.

The algorithm maintains a pseudoflow f and a price function p such that $c_p(v,w) \geq 0$ for every residual arc (v,w). The algorithm consists of repeating the *augmentation step*, described in Figure 5.1, until every excess is zero. Then the current pseudoflow is optimal.

The correctness of this algorithm follows from the observation that the price transformation in the *augment* step preserves complementary slackness and makes the new reduced cost of any minimum-cost path from s to t equal to zero. One iteration of the *augment* step takes time proportional to that required by a single-source shortest path computation on a graph with arcs of non-negative cost. The fastest known strongly polynomial algorithm for this computation is

augment(s,t).
Applicability: $e(s) > 0$ **and** $e(t) < 0$.
Action: For every vertex v, compute $\pi(v)$, the minimum reduced cost of a residual path from s to v. For every vertex v, replace $p(v)$ by $p(v) + \pi(v)$. Move a positive amount of flow from s to t along a path of minimum reduced cost.

Figure 5.1: The augmentation step.

Fredman and Tarjan's implementation [28] of Dijkstra's algorithm, which runs in $O(n \log n + m)$ time. The problem can also be solved in $O(m \log\log C)$ time [84] or $O(n\sqrt{\log C} + m)$ time [2], where C is the maximum arc cost. Since we are mainly interested here in strongly polynomial algorithms, we shall use $O(n \log n + m)$ as our estimate of the time for each *augment* step.

Remark: The only reason to maintain prices in this algorithm is to simplify the minimum-cost path computations by guaranteeing that each such computation is done on a graph all of whose arc costs are non-negative. If prices are not maintained in this way, each shortest path computation takes $O(nm)$ time using the Bellman-Ford algorithm (see e.g. [97]).

5.2 The Edmonds-Karp Algorithm

The overall running time of the augmentation algorithm depends on the number of augmentation steps, which cannot be polynomially bounded without specifying their order more precisely. To impose an efficient order, Edmonds and Karp introduced the idea of capacity scaling, which Orlin [80] in his description of Edmonds-Karp algorithm reinterpreted as excess scaling. We shall present a modification of Orlin's version of the Edmonds-Karp algorithm.

The algorithm maintains a *scaling parameter* Δ such that the flow on every arc is an integer multiple of Δ. For a given pseudoflow f and value of the scaling parameter Δ, we denote the sets of vertices with large excesses and large deficits as follows:

$$\begin{aligned} S_f(\Delta) &= \{v \in V : e_f(v) > \Delta\}; \\ T_f(\Delta) &= \{v \in V : e_f(v) < -\Delta\}. \end{aligned} \tag{5.1}$$

The algorithm consists of a number of scaling phases. During a Δ-scaling phase the residual capacity of every arc is an integer multiple of Δ. The algorithm chooses vertices $s \in S_f(\Delta/2)$, $t \in T_f(\Delta/2)$, performs *augment*(s,t), which sends Δ units of flow along a minimum-cost path from s to t, and repeats such augmentations until either $S_f(\Delta/2)$ or $T_f(\Delta/2)$ is empty. Note that this can create a deficit in place of an excess, and vice versa. Vertices with the new excesses and deficits, however, are not in $S_f(\Delta/2) \bigcup T_f(\Delta/2)$. Then the algorithm halves Δ and begins a new scaling phase. The initial value of Δ is $2^{\lceil \log D \rceil}$. The algorithm terminates after the 1-scaling phase, assuming all supplies are integral. Figure 5.2 provides a detailed description of a phase of the

while $S_f(\Delta/2) \neq \emptyset$ **and** $T_f(\Delta/2) \neq \emptyset$ **do begin**
Choose $s \in S_f(\Delta/2)$ and $t \in T_f(\Delta/2)$. Perform *augment* (s,t), sending Δ units of flow along the augmenting path selected.
end;

Figure 5.2: A phase of the Edmonds-Karp Algorithm.

algorithm.

The following lemma follows from the description of the algorithm.

Lemma 5.2.1 During a Δ-scaling phase, the residual capacity of every arc is an integer multiple of Δ.

Lemma 5.2.2 Let f' be the pseudoflow at the end of a 2Δ-scaling phase. If $S_{f'}(\Delta) = \emptyset$, then there are at most $|S_{f'}(\Delta/2)|$ flow augmentations during the Δ-scaling phase. An analogous statement is true if $T_{f'}(\Delta) = \emptyset$ at the end of a 2Δ-scaling phase.

Proof: By assumption, $S_{f'}(\Delta) = \emptyset$. Therefore, for every vertex $v \in V$, $e_f(v) < \Delta$ during the Δ-scaling phase. This implies that pushing Δ units of flow along a path from $s \in S_f(\Delta/2)$ to $t \in T_f(\Delta/2)$ removes s from $S_f(\Delta/2)$. Note that since $e_f(v) < -\Delta/2$ for $v \in T(\Delta/2)$, vertices with new excesses are not in $S_f(\Delta/2)$. ∎

There are $\lceil \log D \rceil$ phases, each of which consists of up to n single-source shortest path computations on networks with non-negative arc costs. Thus we have the following results for networks with integral supplies:

Theorem 5.2.3 [21] The Edmonds-Karp algorithm solves the transshipment problem in $O((n \log D)(n \log n + m))$ time, and the minimum-cost circulation problem in $O((m \log U)(n \log n + m))$ time.

The excess-scaling idea can be combined with the idea of augmenting along approximately minimum-cost paths rather than exactly minimum-cost paths. Approximately minimum-cost paths can be defined using the ϵ-optimality notion discussed in Chapter 3. An appropriate combination of these ideas yields the *double scaling algorithm* of Ahuja, Goldberg, Orlin, and Tarjan [1]. An implementation of this algorithm based on dynamic trees has a time bound of $O(nm(\log\log U)\log(nC))$ for the minimum-cost circulation problem.

5.3 Strongly Polynomial Algorithms

After Tardos [94] discovered the first strongly polynomial algorithm, Fujishige [29] and independently Orlin [80] developed more efficient strongly polynomial algorithms. Orlin's algorithm [79], that is the main focus of this section, is an extension of these algorithms. These algorithms are based on the method of Section 5.2 and the following dual version of Theorem 4.3.1.

Theorem 5.3.1 [29, 80] Let f be a pseudoflow, and let p be a price function such that the complementary slackness conditions are satisfied and $f(v,w) > \sum_{v:e_f(v)>0} |e_f(v)|$. Then every optimal price function p^* satisfies $c_{p^*}(v,w) = 0$.

Proof: Let p^* be an optimal price function, and let f^* be a corresponding optimal pseudoflow. Assume by way of contradiction that $c_{p^*}(w,v) \neq 0$. Define the pseudoflow f' by $f'(x,y) = f(x,y) - f^*(x,y)$ (*i.e.*, we can obtain an optimal pseudoflow by augmenting f by f'). The Decomposition Theorem (Theorem 1.7.2) implies that f' can be decomposed into flows along a collection of cycles and a collection of paths from vertices with excess to vertices with deficits (both excesses and deficits are with respect to f). Furthermore, the Decomposition Theorem also implies that for any arc (x,y) that is not on one of the cycles, $|f'(x,y)| \leq \sum_{v:e_f(v)>0} e_f(v)$.

The arcs on the cycles of the decomposition are in E_f, and the arcs opposite to the ones on the cycles are in E_{f^*}. This implies that the cycles have zero cost, and that every arc on the cycles must have zero reduced cost with respect to both p and p^*. We have assumed that $c_{p^*}(v,w) \neq 0$; therefore (w,v) cannot be on such a cycle. Thus $f'(w,v) \leq \sum_{v:e_f(v)>0} e_f(v)$ and thus $f^*(v,w) > 0$. But $f^*(v,w) > 0$ and $c_{p^*}(v,w) \neq 0$ contradict the complementary slackness constraint. ∎

The idea behind the strongly polynomial algorithms based on capacity-scaling is to contract the arcs satisfying the condition of Theorem 5.3.1. Let f and p be a pseudoflow and a price function, respectively, and let (v,w) be an arc satisfying the condition of Theorem 5.3.1. The reduced cost function c_p defines an equivalent problem with $c_p(v,w) = 0$. By the above theorem the optimal prices of the vertices v and w are the same. Define a transshipment problem on the network formed by contracting the arc (v,w), with cost function c_p, capacity function u, and demand function d such that the demand of the new vertex is $d(v) + d(w)$.

Theorem 5.3.2 For any optimal price function p^* for the new problem, the price function $p'(r) = p(r) + p^*(r)$ for $r \in V$ (with $p^*(v) = p^*(w)$ defined to be the price of the new vertex) is optimal for the original problem.

Proof: The optimal price function is the optimal solution of a linear program, the linear programming dual of the minimum-cost flow problem. The price function $p' = p + p^*$ is the optimal

Step 1 Run the Edmonds-Karp algorithm for the first $\lceil 2 \log n \rceil$ scaling phases with cost function c_p, starting with $\Delta = \max_{v \in V} |d(v)|$. Let f' be the pseudoflow and let p' be the price function found by the algorithm.
Step 2 Contract every arc (v, w) with $|f(v, w)| > n\Delta$ and update the price function by setting $p(v) \leftarrow p(v) + p'(v)$ for all $v \in V$ (where all vertices $v \in V$ contracted into the same vertex v' of the current network have the same price $p'(v) = p'(v')$).

Figure 5.3: The inner loop of the simple strongly polynomial algorithm.

solution of same linear program with the additional constraint $c(v, w) + p'(v) - p'(w) = 0$. By Theorem 5.3.1, the two linear programs have the same set of optimal solutions. ∎

We shall describe a simple strongly polynomial algorithm based on this idea. It is a variation of Fujishige's algorithm [29], though our description is simplified by using ideas from [79]. One iteration of the algorithm consists of running the Edmonds-Karp algorithm for the first $2 \log n$ scaling phases starting with $\Delta = \max_{v \in V} |d(v)|$ (we cannot find the power of two used by the Edmonds-Karp algorithm in strongly polynomial time), and then contracting all arcs that satisfy the condition of Theorem 5.3.1. The next iteration considers the contracted network. We shall prove that each iteration will contract at least one arc. The algorithm terminates when the current pseudoflow is a feasible solution. Theorem 5.3.2 guarantees that the price function found at this point is optimal. Given an optimal price function, the optimal flow can be found by a single maximum flow computation. (See Figure 5.3 for a more detailed description.)

The following theorem implies that at least one arc is contracted at each iteration of the inner loop.

Theorem 5.3.3 [79] Let f and Δ be the pseudoflow and the scaling parameter at the end of an iteration of the algorithm of Figure 5.3. Then $\sum_{v \in V} |e_f(v)| < n\Delta$. Furthermore, if f is not feasible, then there exists an arc (v, w) with $|f(v, w)| \geq n\Delta$.

Proof: After the Δ-scaling phase either $S_f(\Delta/2)$ or $T_f(\Delta/2)$ is empty. For a pseudoflow f, the excesses sum to zero. Therefore $\sum_{v \in V} |e_f(v)| < n\Delta$. Also, every excess is less than $n\Delta$. After $\lceil 2 \log n \rceil$ scaling phases, $\Delta \leq (\max_{w \in V} |d(w)|)/n^2$. For a vertex v whose demand has the maximum absolute value this implies that

$$|\sum_{w \in V} f(v, w)| \geq |d(v)| - n\Delta \geq n(n-1)\Delta. \tag{5.2}$$

Consequently, at least one arc incident to v carries at least $n\Delta$ flow. ∎

One iteration takes $O(n(n \log n + m) \log n)$ time. Each iteration contracts at least one arc, and therefore there are at most n iterations.

Theorem 5.3.4 The algorithm of Figure 5.3 solves the transshipment problem in $O(n^2 \log n(n \log n + m))$ time, and the minimum-cost circulation problem in $O(m^2 \log n(n \log n + m))$ time.

This simple algorithm wastes a lot of time by throwing away the current flow after each contraction. If there are several vertices with demand close to the maximum then it might make sense to run somewhat more than $2 \log n$ scaling phases. The proof of Theorem 5.3.3 would then apply to more than one vertex, and more than one arc could be contracted in an iteration. This idea does not help much if there are not enough vertices with close-to-maximum demand. On the other hand, by Lemma 5.2.2 the number of shortest path computations during a phase can be bounded in terms of the number of vertices with relatively large demand (it is bounded either by $|S(\Delta/2)|$ or by $|T(\Delta/2)|$). One could try to find a point for stopping the scaling algorithm that balances the number of shortest path computations done and the number of arcs contracted.

Galil and Tardos [36] developed an $O(n^2(n \log + m) \log n)$-time minimum-cost circulation algorithm based on this idea. In fact, the balancing technique of Galil and Tardos [36], together with Orlin's [79] proof of Theorem 5.3.3, can be used to give an $O(n(n \log n + m) \log n)$-time algorithm for the transshipment problem. Instead of pursuing this approach further, however, we shall present a variant of a much simpler algorithm due to Orlin [79] that has same efficiency and does not rely on delicate balancing. This algorithm contracts arcs during the scaling phases, without starting the scaling from scratch after each contraction; the algorithm finds an optimal flow directly, without using an optimal price function. The algorithm is a modification of the Edmonds-Karp algorithm, but runs in $O(n \log n)$ iterations instead of the $O(n \log D)$ iterations of the Edmonds-Karp algorithm.

Orlin's algorithm runs the Edmonds-Karp scaling algorithm starting with $\Delta = \max_{v \in V} |d(v)|$ and contracts all arcs that carry at least $4n\Delta$ flow. The scaling algorithm continues as long as the current pseudoflow is non-zero on some uncontracted arc. When the pseudoflow is zero on every arc, the scaling is restarted by setting Δ to be the maximum absolute value of a demand. Roughly speaking, a vertex v is the start of at most $2 \log n$ shortest path computations during the algorithm. For a vertex v of the original graph, this follows from the fact that $v \notin S_f(\Delta) \cup T_f(\Delta)$ unless $|d(v)| \geq \Delta$, and therefore an arc incident to v will be contracted at most $2 \log n$ scaling phases after v first served as a starting vertex. (See the proof of Theorem 5.3.3.) This may not be true for the contracted vertices, however. A vertex created by a contraction may have very small demand relative to its current excess.

We say that a vertex v is *active* if $v \in S_f(\Delta/2) \cup T_f(\Delta/2)$. A vertex v is *activated* by a Δ-scaling phase if v is not active at the end of the 2Δ-scaling phase, and it becomes active during the Δ-scaling phase (*i.e.*, either it becomes active at the beginning of the phase or the vertex is created by contraction during the phase). We can prove the following lemma using a proof similar to that of Lemma 5.2.2.

Lemma 5.3.5 The number of shortest path computations during a phase is bounded by the number of vertices activated by the phase.

Step 1. Let $\Delta = \max_{v \in V} |d(v)|$. **If** $\Delta = 0$ **then** STOP.
Step 2. while $S_f(\Delta/2) \neq \emptyset$ **and** $T_f(\Delta/2) \neq \emptyset$ **do begin**
Choose $s \in S_f(\Delta/2)$ and $t \in T_f(\Delta/2)$. Perform *augment*(s,t), sending Δ units of flow along the augmenting path selected. Contract every arc (v,w) with $f(v,w) > 4n\Delta$.
end.
Step 3. If f is zero on all uncontracted arcs go to Step 1, otherwise let $\Delta = \Delta/2$ and go to Step 2.

Figure 5.4: Orlin's Algorithm.

Theorem 5.3.6 [79] A vertex can be activated at most $\lceil 2 \log n \rceil + 1$ times before it is contracted.

Proof: A vertex can be activated once due to contraction. When a vertex v is activated for the second time, it must already exist at the end of the previous scaling phase, when it was not active. Let Δ be the scaling parameter in the phase when v is activated for the second time. At the beginning of this phase, $\Delta/2 < |e_f(v)| \leq \Delta$. But $d(v) - e_f(v)$ is an integer multiple of 2Δ. This implies that $\Delta/2 \leq |e_f(v)| \leq |d(v)|$. After $O(\log n)$ more scaling phases the scaling parameter will be less than $|d(v)|/4n^2$. Using an argument analogous to the proof of Theorem 5.3.3 we can conclude that some arc incident to v is contracted. ∎

The previous lemma and theorem bound the number of shortest path computations during the algorithm. All other work done is linear per scaling phase. At least one arc is contracted in each group of $O(\log n)$ scaling phases. Therefore, there are at most $O(n \log n)$ scaling phases.

Theorem 5.3.7 [79] Orlin's algorithm solves the transshipment problem in $O(n \log \min\{n, D\}(n \log n + m))$ time, and the minimum-cost circulation problem in $O(m \log \min\{n, U\}(n \log n + m))$ time.

Remark: In contrast to the simpler strongly polynomial algorithm discussed earlier, Orlin's algorithm constructs an optimal pseudoflow directly, without first constructing an optimal price function. To see this, consider the amount of flow moved during the Δ-scaling phase for some value Δ. Lemma 5.3.5 gives a $2n\Delta$ bound. Suppose an arc is contracted during the Δ-scaling phase. The overall amount of flow that is moved after this contraction can be bounded by $4n\Delta$. Therefore the pseudoflow constructed by the algorithm is feasible in the uncontracted network.

Orlin [80] has observed that capacity scaling ideas can be used to guide pivot selection in the dual network simplex algorithm for the transshipment problem. More recently Orlin, Plotkin and Tardos (personal communication, 1989) have obtained an $O(nm \log \min(n, D))$ bound on the number of pivots based on such a pivot selection strategy.

Chapter 6

The Generalized Flow Problem

6.1 Introduction

The generalized flow problem models the following situation in financial analysis. An investor wants to take advantage of the discrepancies in the prices of securities on different stock exchanges and of currency conversion rates. His objective is to maximize his profit by trading on different exchanges and by converting currencies. The generalized flow problem, considered in this chapter, models the above situation, assuming that a bounded amount of money is available to the investor and that bounded amounts of securities can be traded without affecting the prices. Vertices of the network correspond to different currencies and securities, and arcs correspond to possible transactions.

The generalized flow problem was first considered by Jewell [60]. This problem is very similar to the minimum-cost circulation problem, and several of the early minimum-cost circulation algorithms have been adapted to this problem. The first simple combinatorial algorithms were developed by Jewell [60] and Onaga [78]. These algorithms are not even pseudopolynomial, however, and for real-valued data they need not terminate. Several variations suggested in the early 70's result in algorithms running in finite (but exponential) time. The paper by Truemper [100] contains an excellent summary of these results.

The generalized flow problem is a special case of linear programming, and therefore it can be solved in polynomial time using any general-purpose linear programming algorithm, such as the ellipsoid method [68] and the interior-point algorithms [64, 87]. Kapoor and Vaidya [63] developed techniques to use the structure of the matrices that arise in linear programming formulations of network flow problems to speed up interior-point algorithms. The first two polynomial-time *combinatorial* algorithms were developed by Goldberg, Plotkin, and Tardos [41]. We shall review one of their algorithms in detail and sketch the other one.

The current fastest algorithm for the generalized flow problem, due to Vaidya, is based on an interior-point method for linear programming and runs in $O(n^2m^{1.5}\log(nB))$ time [101]. This algorithm combines the ideas from the paper of Kapoor and Vaidya [63] with the current fastest

linear programming algorithm (which uses fast matrix multiplication). The two combinatorial algorithms due to Goldberg, Plotkin and Tardos [41] run in $O(mn^2(m + n\log n)\log n \log B)$ and $O(m^2n^2 \log n \log^2 B)$ time, respectively. More recently Maggs (personal communication, 1989) improved the latter bound through better balancing to $O\left(n^2m^2 log^2 B \frac{\log n}{\log((n^2 \log n)/m)+\log\log B}\right)$.

6.2 Simple Combinatorial Algorithms

Jewell [60] and Onaga [77] suggested solving the generalized flow problem by using adaptations of augmenting path algorithms for the maximum flow and minimum-cost circulation problems. Theorem 1.6.2 is the basis of Onaga's augmenting path algorithm for the restricted problem. Starting from the zero flow, this algorithm iteratively augments the flow along a flow-generating cycle in the residual graph, thereby increasing the excess at the source. The structure of the restricted problem makes it possible to find the most efficient flow-generating cycle, *i.e.*, the residual flow-generating cycle with highest-gain. The highest-gain cycle consists of an arc (r, s) and a highest-gain simple path from s to r for some vertex r. The highest-gain simple path to r is the shortest path, if the length of each arc (v, w) is defined as $l(v, w) = -\log\gamma(v, w)$. Such a shortest path exists since deleting the arcs entering the source from the residual graph of the initial zero flow yields a graph with no negative cycles. The main observation of Onaga is that if the augmentation is done using the most-efficient flow-generating cycle, then all flow-generating cycles in the residual graph of the resulting generalized flow pass through the source. Onaga's algorithm iteratively augments the generalized flow along the most efficient flow-generating cycle in the residual graph. This algorithm maintains the invariant that every vertex is reachable from s in the current residual graph, a property that can be verified by induction on the number of augmentations.

Consider the special case of a network with two distinguished vertices $s, t \in V$ and with all gains other than $\gamma(t, s)$ and $\gamma(s, t)$ equal to one. The generalized flow problem in this network is equivalent to the maximum flow problem, and Onaga's algorithm specializes to the Ford-Fulkerson maximum flow algorithm. Consequently, the algorithm does not run in finite time.

The next algorithm we describe uses a maximum flow computation as a subroutine. To describe this algorithm, we need to introduce a relabeling technique of Glover and Klingman [37]. This technique can be motivated as follows. Recall the financial analysis interpretation of the generalized flow problem, in which vertices correspond to different securities or currencies, and arcs correspond to possible transactions. Suppose one country decides to change the unit of currency. (For example, Great Britain could decide to introduce the penny as the basic currency unit, instead of the pound, or Italy could decide to erase a couple of 0's at the end of its bills.) This causes an appropriate update of the exchange rates. Some of the capacities change as well (a million $\mathcal{L}$ limit on the $\mathcal{L}$ – DM exchange would now read as a limit of 100 million pennies). It is easy to see that such a relabeling defines an equivalent problem. Such a relabeling can be used, for example, to normalize

local units of currency to current market conditions.

To formally define the *relabeled problem*, let $\mu(v) \in \mathbf{R}^+$ denote the number of old units corresponding to each new unit at vertex $v \in V$. Given a function μ, we shall refer to $\mu(v)$ as the *label* of v.

Definition: For a function $\mu : V \longmapsto \mathbf{R}^+$ and a network $N = (V, E, \gamma, u)$, the *relabeled network* is $N_\mu = (V, E, \gamma_\mu, u_\mu)$, where the *relabeled capacities* and the *relabeled gains* are defined by

$$\begin{aligned} u_\mu(v,w) &= u(v,w)/\mu(v) \\ \gamma_\mu(v,w) &= \gamma(v,w)\mu(v)/\mu(w). \end{aligned}$$

For a generalized pseudoflow g and a labeling μ, the generalized pseudoflow corresponding to g in the relabeled network is $g_\mu(v,w) = g(v,w)/\mu(v)$, the *relabeled residual capacity* is defined by $u_{g,\mu}(v,w) = (u(v,w) - g(v,w))/\mu(v)$, and the *relabeled excess* by $e_{g,\mu}(v) = e_g(v)/\mu(v)$. The corresponding pseudoflows have the same residual graph.

Now we present a *canonical relabeling*. The residual graph of a generalized pseudoflow g can be canonically relabeled if every vertex $v \in V$ is reachable from the source and every flow-generating cycle in the residual graph contains the source. For a vertex $v \in V$, the canonical label $\mu(v)$ is defined to be the gain of a highest-gain simple path from s to v in the residual graph. That is, one new unit corresponds to the amount of flow that can reach the vertex v if one old unit of flow is pushed along a most-efficient simple path in the residual graph from s to v, ignoring capacity restrictions along the path. Observe that in a restricted network, the highest-gain paths from s to each other vertex can be found using any single-source shortest path algorithm.

Theorem 6.2.1 After a canonical relabeling, the following properties hold: Every arc $(v,w) \in E_g$ such that $w \neq s$ has $\gamma_\mu(v,w) \leq 1$; there exists a path from s to every other vertex r in the residual graph with $\gamma_\mu(v,w) = 1$ for all arcs (v,w) on the path; the most efficient flow-generating cycles each consist of an arc $(r,s) \in E_g$ and an (s,r)-path for some $r \in V$, such that $\gamma_\mu(v,w) = 1$ along the path and $\gamma_\mu(r,s) = \max(\gamma_\mu(v,w) : (v,w) \in E_g)$.

Next we describe a simple finite algorithm, due to Truemper [100]. The algorithm, described in Figure 6.1, is a refinement of Onaga's algorithm in which augmentation along all of the maximum gain cycles is done simultaneously. The algorithm maintains a generalized pseudoflow g whose residual graph has every vertex reachable from the source and has all flow-generating cycles passing through the source. The algorithm first canonically relabels the residual graph. Now the highest-gain residual cycles have maximum relabeled gain on arcs entering the source and have a relabeled gain of one on all other arcs. Consider the subgraph induced by arcs with relabeled gain of one and arcs of maximum relabeled gain entering the source. A circulation that maximizes the sum of the flow on the latter arcs can be found by a maximum flow computation. This circulation gives an augmentation of the current generalized flow g. After such an augmentation, all gain cycles in

Step 1 Find μ, a canonical labeling from the source.
If $\gamma_\mu(v,w) \leq 1$ on every arc of the residual graph of the current generalized flow g, then STOP (the current flow is optimal).

Step 2 Let $\alpha = \max_{(v,w)\in E_g} \gamma_\mu(v,w)$. Consider the network G' consisting of all arcs (v,w) with $\gamma_\mu(v,w) = 1$, and all arcs in $A = \{(v,s) | \gamma_\mu(v,s) = \alpha\}$ and their opposites.
Find a (standard) circulation f' in G' that maximizes $\sum_{(v,s)\in A} f'(v,s)$, the flow into s.

Step 3 Let g' be the generalized flow corresponding to f', *i.e*, $g'(v,w) = f'(v,w)$ if $v \neq s$ and $g'(s,v) = -\gamma_\mu(v,s)f'(v,s)$.
Update the current solution by setting $g(v,w) = g(v,w) + g'(v,w)\mu(v)\ \forall (v,w) \in E$.

Figure 6.1: The Inner loop of Truemper's algorithm.

the residual graph pass through the source, and the maximum gain of a flow-generating cycle in the residual graph is decreased. Therefore, this algorithm runs in finite time.

Truemper's algorithm is in some sense an analog of Jewell's [60] minimum-cost flow algorithm, which augments the flow along all of the cheapest augmenting paths at once using a maximum flow subroutine. Using the same network as Zadeh [104] used for the minimum-cost flow problem, Ruhe [89] gave an example on which Truemper's algorithm takes exponential time.

Goldberg, Plotkin, and Tardos [41] gave two algorithms, one that uses a minimum-cost flow computation as a subroutine, and one that builds on ideas from several maximum and minimum-cost flow algorithms. In the next section we describe the first algorithm in detail and very briefly outline the second one.

6.3 Polynomial-Time Combinatorial Algorithms

The main idea of the first polynomial-time algorithm of Goldberg-Plotkin-Tardos is best described by contrasting the algorithm with Truemper's. In each iteration, both algorithms solve a simpler flow problem, and interpret the result as an augmentation in the generalized flow network. Truemper's algorithm is slow because at each iteration it considers only arcs with unit relabeled gain and some of the arcs adjacent to the source, disregarding the rest of the graph completely. The Goldberg-Plotkin-Tardos algorithm, which we shall call *algorithm MCF*, considers all arcs. It assigns a cost $c(v,w) = -\log \gamma_\mu(v,w)$ to each arc (v,w) and solves the resulting minimum-cost circulation problem (disregarding gains).

The *interpretation* of a pseudoflow f is a generalized pseudoflow g, such that $g(v,w) = f(v,w)$ if $f(v,w) \geq 0$ and $g(v,w) = -\gamma_\mu(w,v)f(w,v)$ otherwise. In Truemper's algorithm, the interpretation of a feasible circulation on the subnetwork G' is a feasible generalized flow on the original network. In Algorithm MCF, however, the interpretation of a minimum-cost circulation is a generalized pseudoflow. Arcs of the flow that have a relabeled gain of less than 1 produce vertices with deficits in the interpretation. A connection between a pseudoflow f and its interpretation is given by the

Step 1 Find μ, a canonical labeling of the residual graph of the generalized pseudoflow g. If $\gamma_\mu(v,w) \leq 1$ on every arc of the residual graph and $\forall v \in (V - \{s\}) : e_{g,\mu}(v) = 0$, then HALT (the current flow is optimal).

Step 2 Introduce costs $c(v,w) = -\log \gamma_\mu(v,w)$ on the arcs of the network. Solve the capacitated transshipment problem in the residual relabeled network of G with demands $d(v) = e_{g,\mu}(v)$ for $v \in V - \{s\}$.

Step 3 Let g' be the interpretation of f'. Update the current solution by setting $g(v,w) = g(v,w) + g'(v,w)\mu(v)\ \forall (v,w) \in E$.

Figure 6.2: Inner loop of Algorithm MCF.

following lemma.

Lemma 6.3.1 The residual graphs of a pseudoflow f and its interpretation g as a generalized pseudoflow are the same.

The first iteration of the algorithm solves a minimum-cost circulation problem, and it creates excess at the source and deficits at various other vertices. Each subsequent iteration uses some of the excess at the source to balance the deficits created by the previous iterations by solving a capacitated transshipment problem. More precisely, Algorithm MCF, shown in Figure 6.2, maintains a generalized pseudoflow g in the original (non-relabeled) network, such that the excess at every vertex other than the source is non-positive. The algorithm proceeds in iterations. At each iteration it canonically relabels the residual graph, solves the corresponding capacitated transshipment problem in the relabeled network, and interprets the result as a generalized augmentation.

The most important property of a minimum-cost pseudoflow, for this application, is that its residual graph has no negative cycles. This implies (by Lemma 6.3.1) that the residual graphs of the generalized pseudoflows produced by the algorithm in Step 3 have no flow-generating cycles. The following lemma (analogous to Lemma 1.6.1) implies that this is enough to ensure that the new residual graph can be canonically relabeled.

Lemma 6.3.2 Let g be a generalized pseudoflow in a restricted network. If the residual graph of G_g contains no flow-generating cycles, and the excess at every vertex other than s is non-positive, then every vertex is reachable from s in G_g.

The relabeled gain factors are at most 1 (except for arcs entering the source in the first iteration); therefore the flow computation creates deficits, but no excesses at vertices other than the source. Using the decomposition of pseudoflows (Theorem 1.7.3), one can prove that in each relabeled network there exists a flow satisfying all the deficits. These properties are summarized in the following lemma.

Lemma 6.3.3 For a generalized pseudoflow g that is constructed by Step 3, the following properties hold: The residual graph of g has no flow-generating cycle; canonical relabeling applies to the residual

graph of g; all excesses, except the excess of the source, are non-positive; the pseudoflow required in Step 2 of the next iteration exists.

The algorithm terminates once a generalized flow has been found. Throughout the computation the residual graph contains no flow-generating cycles; therefore (by Theorem 1.6.2) if the algorithm ever finds a generalized flow, then this flow is optimal.

Next we bound the number of iterations of the algorithm. Consider the generalized pseudoflow g existing at the beginning of an iteration. Because there are deficits but no flow-generating cycles in the residual graph, the current excess at the source is an overestimate of the value of the maximum possible excess. It is easy to see that, after a canonical relabeling, the sum of the deficits at all the vertices other than the source is a lower bound on the amount of the overestimate. We use this value, $Def(g,\mu) = \sum_{v \neq s}(-e_{g,\mu}(v))$, as a measure of the proximity of a generalized pseudoflow to an optimal generalized flow. It is not too hard to show that if $Def(g,\mu)$ is smaller than L^{-1} then the algorithm terminates in one more iteration. (Recall that L is the product of denominators of arc gains.)

Theorem 6.3.4 [41] If $Def(g,\mu) < L^{-1}$ before Step 2 of an iteration, then the algorithm produces a generalized flow in Step 3.

An important observation is that the labels μ are monotonically decreasing during the algorithm. The decrease in the labels is related to the price function associated with the minimum-cost pseudoflow computation. Let p' denote the optimal price function associated with the pseudoflow f' found in Step 2. Assume, without loss of generality, that $p'(s) = 0$.

Lemma 6.3.5 For each vertex v, the canonical relabeling in Step 1 of the next iteration decreases the label $\mu(v)$ by at least a factor of $2^{-p'(v)}$.

The key idea of the analysis is to distinguish two cases: Case 1, in which the pseudoflow f' is along "cheap" paths, (*i.e.*, $p'(v)$ is small, say $p'(v) < \log 1.5$, for every $v \in V$); and Case 2, in which there exists a vertex v such that $p'(v)$ is "large" ($\geq \log 1.5$). In the first case, $Def(f,\mu)$ decreases significantly; in the second case, Lemma 6.3.5 implies that at least one of the labels decreases significantly. The label of a vertex v is the gain of the current most efficient path from s to v. This limits the number of times Case 2 can occur. Theorem 6.3.4 can be used to guarantee that Case 1 cannot occur too often. The following lemma provides a tool for estimating the total deficit created when interpreting a minimum-cost pseudoflow as a generalized pseudoflow.

Lemma 6.3.6 Let f' be a pseudoflow along a simple path P from s to some other vertex v that satisfies one unit of deficit at v. Let g' be the interpretation of f' as a generalized pseudoflow. Assume that all relabeled gains along the path p are at most 1, and denote them by $\gamma_1, \ldots, \gamma_k$. Then after

augmenting by g', the unit of deficit at v is replaced by deficits on vertices along P that sum up to at most $(\prod_{1\leq i\leq k}\gamma_i)^{-1}-1$.

Proof: The deficit created at the ith vertex of the path is $(1-\gamma_i)$, for $i=1,\ldots,k$. Using the assumption that the gain factors along the path are at most 1, the sum of the deficits can be bounded by

$$\sum_{i=1}^{k}(1-\gamma_i)\leq\sum_{i=1}^{k}\frac{1-\gamma_i}{\prod_{j=1}^{i}\gamma_j}=\frac{1}{\prod_{i=1}^{k}\gamma_i}-1.$$

∎

This lemma can be used to bound the value of $Def(g,\mu)$ after an application of Step 3. Let p' be an optimal price function associated with the pseudoflow f', such that $p'(s)=0$. Let $\beta=\max_{v\in V}p'(v)$ be the maximum price. Using the pseudoflow decomposition theorem (Theorem 1.7.2), one can show that a minimum-cost pseudoflow f' can be decomposed into flows along paths from the source s to the other vertices and cycles in the residual graph of g, such that the cost of the cycles is zero and the cost of each path ending at a vertex v is at most $p'(v)$. To bound the amount of deficit created by the interpretation, we shall consider the interpretation of the flow on the cycles and the paths one-by-one. Since the cycles consists of zero-cost arcs only, the interpretation of the flow along these cycles does not create deficits. When interpreting flow along a path from s to v, the deficit at v is replaced by deficits that sum to at most $2^{p'(v)}-1\leq 2^{\beta}-1$ times the deficit at v satisfied by this path. This proves the following lemma.

Lemma 6.3.7 The value of $Def(g,\mu)$ after an application of Step 3 can be bounded by $2^{\beta}-1$ times its value before the step. In particular, if $p'(v)<\log 1.5$ for every vertex v, then $Def(g,\mu)$ decreases by a factor of 2.

The remaining difficulty is the fact that the function $Def(g,\mu)$ can increase, both when Case 2 applies in Step 3 and due to the relabeling in Step 1 (changing the currency unit from $\mathcal{L}$ to penny increases a 5 $\mathcal{L}$ deficit to 500 pennies). The increase in $Def(g,\mu)$ can be related to the decrease in the labels, however. The deficit at a vertex increases in Step 1 by a factor of α if and only if the label of this vertex decreases by the same factor. The increase in $Def(g,\mu)$ during Step 3 is bounded by 2^{β}, where $\beta=\max p'(v)$, by Lemma 6.3.7. By Lemma 6.3.5, this means that $\mu(v)$ for some vertex v decreases by at least β during Step 1 of the next iteration. Hence, in both cases, if $Def(g,\mu)$ increases by α during a step, then there exists a vertex v for which $\mu(v)$ decreases by at least α during the next execution of Step 1. Let $T\leq B^n$ denote an upper bound on the gain of any simple path. The label of a vertex v is the gain of a simple path from s to v in the residual graph. Therefore the labels are at least T^{-1} and at most T, and the label of a vertex cannot decrease by more than a factor of T^2 during the algorithm. This gives the following lemma.

Lemma 6.3.8 The growth of the function $Def(g, \mu)$ throughout an execution of the algorithm is bounded by a factor of $T^{O(n)} = B^{O(n^2)}$.

Theorem 6.3.9 [41] The algorithm terminates in $O(n^2 \log B)$ iterations.

Proof: The label of any vertex v is the gain of a simple path from s to v, and it is monotonically decreasing during the algorithm. Therefore, it cannot decrease by a constant factor more than $O(\log T)$ times for any given vertex, and Case 2 cannot occur more than $O(n \log T)$ times. When Case 1 applies, the value of $Def(g, \mu)$ decreases by a factor of 2. The value of $Def(g, \mu)$ is at most $O(nBT)$ after the first iteration; and, by Theorem 6.3.4, the algorithm terminates when this value decreases below L^{-1}. Lemma 6.3.8 limits the increase of $Def(g, \mu)$ to $T^{O(n)}$ during the algorithm. Hence, Case 1 cannot occur more than $O(\log(nBT) + n \log T + \log L) = O(n^2 \log B)$ times. ∎

To get a bound on the running time, one has to decide which algorithm to use for computing the minimum-cost pseudoflow in Step 2. The best choice is Orlin's $O(m(m + n \log n) \log n)$ time algorithm, discussed in Chapter 5.

Theorem 6.3.10 [41] Algorithm MCF solves the generalized flow problem using at most $O(n^2 m(m + n \log n) \log n \log B)$ arithmetic operations on numbers whose size is bounded by $O(m \log B)$.

We conclude this section with a brief discussion of the other algorithm of [41]. The algorithm is based on ideas from two flow algorithms: the cycle-canceling algorithm of Goldberg and Tarjan [46] described in Chapter 4, and the fat-path maximum flow algorithm of Edmonds and Karp [21], which finds the maximum flow in a network by repeatedly augmenting along a highest-capacity residual path.

Each iteration of the algorithm starts with cycle-canceling. Canceling a flow-generating cycle in a generalized flow network creates an excess at some vertex of the cycle. If we cancel cycles other than the most efficient ones, the residual graph of the resulting generalized pseudoflow will have flow-generating cycles that do not contain the source. The algorithm cancels flow-generating cycles by an adaptation of the Goldberg-Tarjan algorithm. Using the dynamic tree data structure, this phase can be implemented to run in $O(n^2 m \log n \log B)$ time. The resulting excesses at vertices other than the source are moved to the source along augmenting paths in the second phase of the algorithm.

Consider a generalized pseudoflow that has non-negative excesses and whose residual graph has no flow-generating cycles. The key idea of the second phase is to search for augmenting paths from vertices with excess to the source that result in a significant increase in the excess of the source. The algorithm maintains a scaling parameter Δ such that the maximum excess at the source is at most $2m\Delta$ more than the current excess. It looks for an augmenting path that can increase the

excess of the source by at least Δ. If the residual graph of the current generalized pseudoflow has no flow-generating cycles, then one can find a sequence of such paths such that, after augmenting the generalized flow along these paths, the maximum excess at the source is at most $m\Delta$ more than the current excess. This phase can be implemented in $O(m(m + n \log n))$ time by a sequence of $O(m)$ single-source shortest path computations on graphs with arcs of non-negative cost. Now Δ is divided by two and a new iteration is started. After $O(m \log B)$ iterations, when the excess at the source is very close to the optimum value, Truemper's algorithm can be applied to bring all of the remaining excesses to the source.

Maggs (personal communication, 1989) has observed that this algorithm can be improved through better balancing of the two phases. Dividing Δ by a factor of 2^α after each iteration decreases the number of iterations by a factor of α and increases the time required for the second phase by a factor of 2^α. The parameter α is chosen so that the time required for the two phases is balanced. The resulting algorithm runs in $O\left(n^2 m^2 \log^2 B \frac{\log n}{\log((n^2 \log n)/m) + \log\log B}\right)$ time.

Bibliography

[1] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. *Finding Minimum-Cost Flows by Double Scaling.* Technical Report CS-TR-164-88, Department of Computer Science, Princeton University, 1988.

[2] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. *Faster Algorithms for the Shortest Path Problem.* Technical Report CS-TR-154-88, Department of Computer Science, Princeton University, 1987.

[3] R. K. Ahuja and J. B. Orlin. A Fast and Simple Algorithm for the Maximum Flow Problem. 1987. Sloan Working Paper 1905-87, Sloan School of Management, M.I.T.

[4] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. *Improved Time Bounds for the Maximum Flow Problem.* Technical Report CS-TR-118-87, Department of Computer Science, Princeton University, 1987. (SIAM J. Comput., to appear).

[5] B. Awerbuch. Personal Communication. 1985. Laboratory for Computer Science, M.I.T.

[6] L. M. Balinski. Signature Methods for the Assignment Problem. *Oper. Res.*, 33:527–536, 1985.

[7] F. Barahona and É. Tardos. Note on Weintraub's Minimum Cost Circulation Algorithm. *SIAM J. Comput.*, to appear.

[8] C. Berge. *Graphs and Hypergraphs.* North Holland, 1973.

[9] D. P. Bertsekas. A Distributed Algorithm for the Assignment Problem. 1979. Unpublished Manuscript, Lab. for Decision Systems, M.I.T.

[10] D. P. Bertsekas. A Distributed Asynchronous Relaxation Algorithm for the Assignment Problem. In *Proc. 24th IEEE Conference on Decision and Control, Ft. Lauderdale, FL*, 1985.

[11] D. P. Bertsekas. *Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems.* Technical Report LIDS-P-1986, Lab. for Decision Systems, M.I.T., 1986.

[12] D. P. Bertsekas and J. Eckstein. Dual Coordinate Step Methods for Linear Network Flow Problems. *Math. Programming*, 42:203–243, 1988.

[13] 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.

[14] R. G. Busacker and P. J. Gowen. *A Procedure for Determinimg a Family of Minimal-Cost Network Flow Patterns.* Technical Report 15, O.R.O., 1961.

[15] R. G. Busacker and T. L. Saaty. *Finite Graphs and Networks: An Introduction with Applications.* McGraw-Hill, New York, NY., 1965.

[16] J. Cheriyan and S.N. Maheshwari. Analysis of a Preflow Push Algorithm for Maximum Netwrok Flow. 1987. Unpublished manuscript, Department of Computer Science and Engineering, Indian Institute of Technology, New Deli, India.

[17] 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).

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

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

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

[21] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

[22] G. M. Engel and H. Schneider. Diagonal Similarity and Equivalence for Matrices Over Groups with 0. *Czech. Math. J.*, 25:389–403, 1975.

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

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

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

[26] L. R. Ford, Jr. and D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Math.*, 8:399–404, 1956.

[27] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[28] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[29] S. Fujishige. A Capacity-rounding Algorithm for the Minimum-Cost Circulation Problem: a Dual Framework of the Tardos Algorithm. *Math. Prog.*, 35:298–308, 1986.

[30] D. R. Fulkerson. An Out-of-Kilter Method for Minimal Cost Flow Problems. *SIAM J. Appl. Math*, 9:18–27, 1961.

[31] H. N. Gabow. Scaling Algorithms for Network Problems. *J. of Comp. and Sys. Sci.*, 31:148–168, 1985.

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

[33] H. N. Gabow and R. E. Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, to appear.

[34] Z. Galil. An $O(V^{5/3}E^{2/3})$ Algorithm for the Maximal Flow Problem. *Acta Informatica*, 14:221–242, 1980.

[35] Z. Galil and A. Naamad. An $O(EV\log^2 V)$ Algorithm for the Maximal Flow Problem. *J. Comput. System Sci.*, 21:203–217, 1980.

[36] Z. Galil and É. Tardos. An $O(n^2(m+n\log n)\log n)$ Minimum Cost Flow Algorithm. *J. Assoc. Comput. Mach.*, 35:374–386, 1988.

[37] F. Glover and D. Klingman. On the Equivalence of Some Generalized Network Problems to Pure Network Problems. *Math. Programming*, 4:269–278, 1973.

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

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

[40] A. V. Goldberg, M. D. Grigoriadis, and R. E. Tarjan. *Efficiency of the Network Simplex Algorithm for the Maximum Flow Problem.* Technical Report LCSR-TR-117, Laboratory for Computer Science Research, Department of Computer Science, Rutgers University, 1988.

[41] A. V. Goldberg, S. A. Plotkin, and É. Tardos. *Combinatorial Algorithms for the Generalized Circulation Problem.* Technical Report STAN-CS-88-1209, Stanford University, 1988. (Also available as Technical Memorandum MIT/LCS/TM-358, Laboratory for Computer Science, M.I.T., 1988.).

[42] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-Time Parallel Algorithms for Matching and Related Problems. In *Proc. 29th IEEE Symp. on Found. of Comp. Sci.*, pages 174–185, 1988.

[43] 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.

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

[45] A. V. Goldberg and R. E. Tarjan. *A Parallel Algorithm for Finding a Blocking Flow in an Acyclic Network.* Technical Report STAN-CS-88-1228, Department of Computer Science, Stanford University, 1988.

[46] A. V. Goldberg and R. E. Tarjan. *Finding Minimum-Cost Circulations by Canceling Negative Cycles.* Technical Report MIT/LCS/TM-334, Laboratory for Computer Science, M.I.T., 1987. (Also available as Technical Report CS-TR 107-87, Department of Computer Science, Princeton University).

[47] A. V. Goldberg and R. E. Tarjan. *Finding Minimum-Cost Circulations by Successive Approximation.* Technical Report MIT/LCS/TM-333, Laboratory for Computer Science, M.I.T., 1987. (Math. of Oper. Res., to appear).

[48] A. V. Goldberg and R. E. Tarjan. Solving Minimum-Cost Flow Problems by Successive Approximation. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 7–18, 1987.

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

[50] D. Goldfarb and J. Hao. *A Primal Simplex Algorithm that Solves the Maximum Flow Problem in at most nm Pivots and $O(n^2m)$ Time.* Technical Report, Department of IE and OR, Columbia University, 1988.

[51] M. Gondran and M. Minoux. *Graphs and Algorithms.* Wiley, 1984.

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

[53] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization.* Springer Verlag, 1988.

[54] R. Hassin. Maximum Flows in (s,t) Planar Networks. *Information Processing Let.*, 13:107–108, 1981.

[55] R. Hassin and D. B. Johnson. An $O(n \log^2 n)$ Algorithm for Maximum Flow in Undirected Planar Network. *SIAM J. Comput.*, 14:612–624, 1985.

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

[57] M. Iri. A New Method of Solving Transportation-Network Problems. *J. Op. Res. Soc. Japan*, 3:27–87, 1960.

[58] A. Itai and Y. Shiloach. Maximum Flow in Planar Network. *SIAM J. Comput.*, 8:135–150, 1979.

[59] P. A. Jensen and J. W. Barnes. *Network Flow Programming.* J. Wiley & Sons, 1980.

[60] W. S. Jewell. *Optimal Flow through Networks.* Technical Report 8, M.I.T., 1958.

[61] W. S. Jewell. Optimal Flow through Networks with Gains. *Oper. Res.*, 10:476–499, 1962.

[62] D. B. Johnson and S. Venkatesan. Using Divide and Conquer to Find Flows in Directed Planar Netwroks in $O(n^{1.5} \log n)$ Time. In *Proc. 20th Annual Allerton Conf. on Communication, Control, and Computing*, pages 898–905, 1982.

[63] 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.

[64] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4:373–395, 1984.

[65] R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Math.*, 23:309–311, 1978.

[66] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a Maximum Matching is in Random NC. *Combinatorica*, 6:35–48, 1986.

[67] A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dok.*, 15:434–437, 1974.

[68] L. G. Khachian. Polynomial Algorithms in Linear Programming. *Zhurnal Vychislitelnoi Matematiki i Matematicheskoi Fiziki*, 20:53–72, 1980.

[69] M. Klein. A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems. *Management Science*, 14:205–220, 1967.

[70] D. König. *Theorie der Endlichen und Unendlichen Graphen.* Chelsea Publishing Co., New York, 1950.

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

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

[73] V. M. Malhotra, M. Pramodh Kumar, and S. N. Maheshwari. An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks. *Information Processing Let.*, 7:277–278, 1978.

[74] G. L. Miller and J. Naor. Flow in Planar Graphs with Multiple Sources and Sinks. 1989. Unpublished Manuscript, Computer Science Department, Stanford University, Stanford, CA.

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

[76] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 105–131, 1987.

[77] K. Onaga. Dynamic Programming of Optimum Flows in Lossy Communication Nets. *IEEE Trans. Circuit Theory*, 13:308–327, 1966.

[78] K. Onaga. Optimal Flows in General Communication Networks. *J. Franklin Inst.*, 283:308–327, 1967.

[79] J. B. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 377–387, 1988.

[80] 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, 1984.

[81] J. B. Orlin. On the Simplex Algorithm for Networks and Generalized Networks. *Math. Prog. Studies*, 24:166–178, 1985.

[82] J. B. Orlin and R. K. Ahuja. New Distance-Directed Algorithms for Maximum-Flow and Parametric Maximum-Flow Problems. 1987. Sloan Working Paper 1908-87, Sloan School of Management, M.I.T.

[83] J. B. Orlin and R. K. Ahuja. New Scaling Algorithms for Assignment and Minimum Cycle Mean Problems. 1988. Sloan Working Paper 2019-88, Sloan School of Management, M.I.T.

[84] P. Van Emde Boas and R. Kaas and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

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

[86] J. H. Reif. Minimum *s-t* Cut of a Planar Undirected Network in $O(n \log^2 n)$ Time. *SIAM J. Comput.*, 12:71–81, 1983.

[87] J. Renegar. A Polynomial Time Algorithm, Based on Newton's Method, for Linear Programming. *Mathematical Programming*, 40:59–94, 1988.

[88] H. Röck. Scaling Techniques for Minimal Cost Network Flows. In U. Pape, editor, *Discrete Structures and Algorithms*, pages 181–191, Carl Hansen, Münich, 1980.

[89] G. Ruhe. Parametric Maximal Flows in Generailed Networks – Complexity and Algorithms. *Optimization*, 19:235–251, 1988.

[90] 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.

[91] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ Parallel Max-Flow Algorithm. *J. Algorithms*, 3:128–146, 1982.

[92] 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.

[93] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *J. Comput. System Sci.*, 26:362–391, 1983.

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

[95] R. E. Tarjan. A Simple Version of Karzanov's Blocking Flow Algorithm. *Operations Research Letters*, 2:265–268, 1984.

[96] R. E. Tarjan. Amortized Computational Complexity. *SIAM J. Alg. Disc. Math.*, 6:306–318, 1985.

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

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

[99] N. Tomizawa. On Some Techniques Useful for Solution of Transportation Networks Problems. *Networks*, 1:173–194, 1972.

[100] K. Truemper. On Max Flows with Gains and Pure Min-Cost Flows. *SIAM J. Appl. Math.*, 32:450–456, 1977.

[101] P. M. Vaidya. Speeding up Linear Programming Using Fast Matrix Multiplication. 1989. Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ.

[102] A. Weintraub. A Primal Algorithm to Solve Network Flow Problems with Convex Costs. *Management Science*, 21:87–97, 1974.

[103] M. A. Yakovleva. A Problem on Minimum Transportation Cost. In V. S. Nemchinov, editor, *Applications of Mathematics in Economic Research*, pages 390–399, Izdat. Social'no-Ekon. Lit., Moscow, 1959.

[104] N. Zadeh. A Bad Network Problem for the Simplex Method and Other Minimum Cost Flow Problems. *Mathematical Programming*, 5:255–266, 1973.