JOURNAL OF OPTIMIZATION THEORY AND APPLICATIONS: Vol. 87, No. 1, pp. 69–101, OCTOBER 1995

An Auction Algorithm for the Max-Flow Problem[1,2]

D. P. BERTSEKAS[3]

Communicated by P. Tseng

Abstract. We propose a new algorithm for the max-flow problem. It consists of a sequence of augmentations along paths constructed by an auction-like algorithm. These paths are not necessarily shortest, that is, they need not contain a minimum number of arcs. However, they can be found typically with much less computation than the shortest augmenting paths used by competing methods. Our algorithm outperforms these latter methods as well as state-of-the-art preflow-push algorithms by a very large margin in tests with standard randomly generated problems.

Key Words. Network optimization, max-flow problems, auction algorithms, preflow-push algorithms.

1. Introduction

In this paper, we propose a new algorithm for the classical max-flow problem, where we are given a directed graph $(\mathcal{N}, \mathcal{A})$, and we want to push a maximum amount of flow from a source node 1 to a sink node N, subject to the constraint that the flow of each arc $(i,j) \in \mathcal{A}$ should lie in an interval $[0, c_{ij}]$, where c_{ij} is a given positive scalar, called the capacity of (i,j); here, the number of nodes is N and the nodes are denoted $1, 2, \ldots, N$. The number of arcs is A and, to facilitate the presentation, we assume that there is at most one arc (i,j) starting at i and ending at j, so that we can unambiguously refer to an arc as (i,j). A flow vector

$$x = \{x_{ij} \mid (i,j) \in \mathcal{A}\}$$

[1]This paper is a substantially revised version of Ref. 1.

[2]Many thanks are due to David Castanon and Paul Tseng for several helpful comments. The suggestions of the referees were also appreciated. David Castanon, Lakis Polymenakos, and Won-Jong Kim helped with some of the computational experimentation. This research was supported by NSF under Grant CCR-9103804.

[3]Professor, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.

0022-3239/95/1000-0069$07.50/0

is said to be capacity feasible if

$$0 \leq x_{ij} \leq c_{ij}, \qquad \text{for all } (i,j) \in \mathcal{A}.$$

The associated surplus of each node is defined by

$$g_i = \sum_{\{j \mid (j,i) \in \mathcal{A}\}} x_{ji} - \sum_{\{j \mid (i,j) \in \mathcal{A}\}} x_{ij}, \qquad \forall i \in \mathcal{N}. \tag{1}$$

The flow vector is said to be feasible if it is capacity feasible and the node surpluses satisfy

$$g_i = 0, \qquad \forall i \in \mathcal{N}, i \neq 1, i \neq N. \tag{2}$$

The problem is to find a feasible flow such that g_N is maximized.

The classical approach to the max-flow problem is the Ford–Fulkerson algorithm (Ref. 2), which consists of successive augmentations; it moves flow sequentially from the source to the sink along augmenting paths, until a saturated cut separating the source and the sink is created. In its original form, this algorithm had two drawbacks:

(a) If the augmenting paths are arbitrarily constructed, the number of augmentations can be very large. In fact, if the arc capacities are irrational, the algorithm may fail to terminate (see, e.g., Refs. 3–5).

(b) No mechanism is provided to pass helpful information from one augmenting path construction to the next.

These two drawbacks have been addressed by much subsequent research. The traditional approach to keep the number of augmentations small is to ensure that the augmenting paths are shortest, in the sense that they contain the smallest possible number of arcs. In fact, all polynomial augmenting path methods of which we are aware use this approach. The simplest way to construct the shortest augmenting paths is to use a breadth-first search method, leading to an $O(NA^2)$ running time (Ref. 6). In order to reuse information from one shortest augmenting path construction to the next, the idea of a layered network implementation was also suggested (Ref. 7) and resulted in an $O(N^2A)$ running time.

The algorithm of this paper is of the Ford–Fulkerson type, but does not use shortest augmenting paths. Instead, it constructs possibly nonshortest augmenting paths using the ideas of the auction algorithm for the assignment problem (Refs. 5, 8). In particular, our path construction algorithm is obtained by converting the path construction problem to a special type of unweighted matching problem, applying the auction algorithm, and streamlining the computations. A key feature here is that the price mechanism of the auction algorithm is used to pass valuable information from one augmenting path construction to the next.

Another relevant class of max-flow algorithms is the class of preflow-push methods, which originated with the work of Refs. 9, 10 and has been the subject of much recent development (Refs. 11–17). These methods move flow along single-arc paths, and they share with the auction algorithm the idea of using a price mechanism (within this context, prices are also called labels). This connection is not accidental, and in fact it is shown in Ref. 18 that a generic preflow-push method for the max-flow problem (Ref. 12) can be derived as a special case of the auction algorithm for the assignment problem, using the reformulation of the max-flow problem as an assignment problem. Preflow-push methods have excellent theoretical worst-case complexity [$O(N^2A^{1/2})$ with relatively simple implementation (Ref. 15), and even better through the use of sophisticated but somewhat impractical data structures]. On the basis of some recent studies (Refs. 16, 17, 19, 20), they are reputed to be the fastest in practice, when appropriately implemented.

Our algorithm has an $O(N^2A)$ worst-case running time, but according to our experiments, it is substantially faster than both shortest augmenting path and preflow-push methods. There is a twofold explanation for this. First, the auction algorithm solves simpler path construction problems than the competing shortest augmenting path methods, while at the same time it passes useful price information from one path construction to the next. Second, because flow changes take place over multiple-arc paths, the phenomenon of ping-ponging of flow between pairs of nodes that is characteristic of preflow-push methods is largely avoided. Indeed, our experiments show that the number of arc flow changes required to solve the problem is generally far smaller in our method than in preflow-push methods.

The paper is organized as follows. In Section 2, we describe our auction algorithm for path construction. In Section 3, we embed the path construction algorithm of Section 2 within a sequential augmentation framework to obtain our main max-flow algorithm. We establish the validity of the algorithm, and we show that its running time is $O(N^2A)$. Efficient implementation is very important for the success of our algorithm, and in Section 4 we outline a number of variations that can improve its performance. In Section 5, we present computational results with standard randomly generated problems. These results show that our algorithm outperforms state-of-the-art preflow-push methods by a very large margin under identical test conditions. Finally in the appendix, we describe briefly how our path construction method can be viewed as an application of the auction algorithm for the assignment problem to a special type of unweighted matching problem.

2. Path Construction Algorithms

In this section, we describe a method for finding a path between two nodes of a graph. This method lies at the heart of our max-flow algorithm,

which will be presented in the next section. We give two versions of the algorithm. The first is simple and easy to understand. The second is a more complex variation of the first, but is apparently more efficient in practice. We first introduce some terminology that is common to all sections of this paper.

Consider the directed graph $(\mathcal{N}, \mathcal{A})$ given in the introduction. The set of arcs outgoing from node i is denoted by $A(i)$, and the corresponding set of nodes $\{j \mid (i,j) \in A(i)\}$ is denoted by $N(i)$. A path P is a sequence of nodes $(n_1, n_2, \ldots, n_t)$ with $t \geq 2$, and a corresponding sequence of $t-1$ arcs such that the ith arc in the sequence is either (n_i, n_{i+1}), in which case it is called a forward arc of the path, or (n_{i+1}, n_i), in which case it is called a backward arc of the path. Node n_1 is called the start node of P, and node n_t is called the terminal node of P. By slight abuse of terminology, we consider $P = (n_1)$ to be a path, in which case n_1 is both the start and the terminal node of P. For $i = 2, \ldots, t$, the node n_{i-1} is called the predecessor of n_i, and is denoted by $\text{pred}(n_i)$. We denote by P^+ and P^- the sets of forward and backward arcs of P, respectively. The path P is said to be forward if all its arcs are forward. The path P is said to be simple if it contains no cycles, that is, if the nodes $n_1, \ldots, n_t$ are distinct. The length of a path is the number of its arcs; all future references to shortest paths are with respect to this length. All paths in this section will be forward paths. The paths to be considered in the context of the max-flow problem, starting with the next section, may contain both forward and backward arcs.

The following algorithm aims at finding a simple forward path that starts at a given node n_1 and ends at node N. It maintains a simple forward path $P = (n_1, \ldots, n_t)$ and a set of integer node prices satisfying

$$p(i) \leq p(j) + 1, \qquad \forall (i,j) \in \mathcal{A}, \tag{3}$$

$$p(n_1) < N, \qquad p(N) = 0, \tag{4}$$

$$p(i) \geq p(j), \qquad \forall (i,j) \in P. \tag{5}$$

The conditions (3) and (5) are related to the ϵ-complementary slackness conditions of the auction algorithm (see the appendix); here, $\epsilon = 1$.

The algorithm is motivated by the max-flow context, where the objective is not to find a single path, but rather to find a sequence of paths each in a graph that differs slightly from its predecessor. Within this context, prices are helpful in guiding the search for new paths. Loosely speaking, prices are modified by the algorithm in a way that the desired paths have an approximate downhill direction, that is, they proceed from high price nodes to low price nodes. Thus, if a set of prices is roughly appropriate for guiding the search for a path in a given graph, it is also roughly appropriate for guiding the search for a path in a slightly different graph.

At the start of the algorithm, we require that $P=(n_1)$, and that the vector $p=(p(1), \ldots, p(N))$ is such that Eqs. (3) and (4) hold. The path P is modified repeatedly using the following two operations:

(a) A contraction of P, which deletes the last arc of P, that is, replaces the path $P=(n_1, \ldots, n_t)$ by the path $P=(n_1, \ldots, n_{t-1})$. In the degenerate case where $P=(n_1)$, a contraction leaves P unchanged.

(b) An extension of P, which adds to P an arc outgoing from its terminal node, that is, replaces the path $P=(n_1, \ldots, n_t)$ by a path $P=(n_1, \ldots, n_t, n_{t+1})$, where (n_t, n_{t+1}) is an arc.

The prices $p(i)$ may also be increased in the course of the algorithm so that, together with P, they satisfy the conditions (3)–(5). A contraction always involves a price increase of the terminal node n_t. An extension may or may not involve such a price increase. An extension of P is always done to a neighbor of n_t that has minimal price. The algorithm terminates if either node N becomes the terminal node of P (then, P is the desired path), or else $p(n_1) \geq N$ [in view of $p(N)=0$ and $p(i) \leq p(j)+1$ for all arcs (i,j) as per Eqs. (3) and (4), this means that there is no forward path from n_1 to N].

Algorithm A1. Path Construction Algorithm. Set $P=(n_1)$, and select p such that Eqs. (3) and (4) hold.

Step 1. Check for Contraction or Extension. Let n_t be the terminal node of P. If the set $N(n_t)$ is empty, set $p(n_t)=N$ and go to Step 3. Otherwise, find a node in $N(n_t)$ with minimal price and denote it $\text{succ}(n_t)$,

$$\text{succ}(n_t)=\arg \min_{j \in N(n_t)} p(j). \tag{6}$$

Set

$$p(n_t)=p(\text{succ}(n_t))+1. \tag{7}$$

If $n_t=n_1$ and $p(n_1)<N$, or if

$$n_t \neq n_1 \text{ and } p(\text{pred}(n_t))>p(\text{succ}(n_t)), \tag{8}$$

go to Step 2; otherwise, go to Step 3.

Step 2. Extend Path. Extend P by node $\text{succ}(n_t)$ and the corresponding arc of $A(n_t)$. If $\text{succ}(n_t)=N$, terminate the algorithm; otherwise, go to Step 1.

Step 3. Contract Path. If $P=(n_1)$ and $p(n_1) \geq N$, terminate the algorithm; otherwise, contract P and go to Step 1.

We note that maintaining a path that is extended or contracted at each iteration, while maintaining a price vector that satisfies complementary slackness conditions, is a central feature of the auction algorithm for shortest paths (Refs. 5, 21) and its embedding in a sequential shortest path algorithm for the minimum cost flow problem (Ref. 22). However, as mentioned earlier, our path construction algorithm does not necessarily generate a shortest path. Instead, we show in the appendix that it just solves a special type of unweighted matching problem by means of the auction algorithm.

In the special case where all the initial prices are zero and there is a path from each node to N, by tracing the steps, it can be seen that the algorithm will work like depth-first search, raising to 1 the price of the nodes of some path from n_1 to N in a sequence of extensions with no intervening contractions. More generally, the algorithm terminates without performing any contractions if the initial prices satisfy $p(i) \geq p(j)$ for all arcs (i, j) and there is a path from each node to N.

We make the following observations:

(i) The prices remain integer throughout the algorithm [cf. Eq. (7)].

(ii) The conditions (3)–(5) are satisfied each time Step 1 is entered. The proof is by induction. These conditions hold initially by assumption. Condition (4) is maintained by the algorithm, since we have termination as soon as $p(n_1) \geq N$. To verify conditions (3) and (5), we note that only the price of n_t can change in Step 1, and by Eqs. (6) and (7), this price change maintains condition (3) for all arcs, and condition (5) for all arcs of P, except possibly for the arc $(\text{pred}(n_t), n_t)$ in the case of an extension with the condition

$$p(\text{pred}(n_t)) > p(\text{succ}(n_t))$$

holding. In the latter case, we must have

$$p(\text{pred}(n_t)) \geq p(\text{succ}(n_t)) + 1,$$

because the prices are integer, so by Eq. (7), we have

$$p(\text{pred}(n_t)) \geq p(n_t)$$

at the next entry to Step 1. This completes the induction.

(iii) A contraction is always accompanied by a price increase. Indeed by Eq. (5), which was just established, upon entering Step 1 with $n_t \neq n_1$, we have

$$p(n_t) \leq p(\text{pred}(n_t)),$$

and to perform a contraction, we must have

$$p(\text{pred}(n_t)) \leq p(\text{succ}(n_t)).$$

Hence,

$$p(n_t) \leq p(\text{succ}(n_t)),$$

implying by Eq. (7) that $p(n_t)$ must be increased to the level $p(\text{succ}(n_t))+1$. It can be seen, however, by example, that an extension may or may not be accompanied by a price increase.

(iv) Upon return to Step 1 following an extension, the terminal node n_t satisfies [cf. Eq. (7)]

$$p(\text{pred}(n_t)) = p(n_t) + 1. \tag{9}$$

This, together with the condition $p(i) \geq p(j)$ for all $(i,j) \in P$ [cf. Eq. (5)], implies that the path P will not be extended to a node that already belongs to P. Thus P remains a simple path throughout the algorithm.

To facilitate the presentation, let us introduce some additional terminology. For a given integer price vector p, we say that an arc (i,j) is uphill if $p(i) < p(j)$, downhill if $p(i) \geq p(j)$, and strictly downhill if $p(i) = p(j)+1$. The following proposition summarizes the conclusions of the preceding discussion and establishes the termination properties of the algorithm.

Proposition 2.1.

(a) Throughout the algorithm, the prices satisfy the conditions (3) and (4), the path P is simple, its arcs are downhill, and following an extension, the last arc of P is strictly downhill.

(b) If there exists a forward path from n_1 to N, the algorithm terminates via Step 2 with such a path. Otherwise, the algorithm terminates via Step 3.

Proof. Part (a) was established above, so we prove part (b). We first note that the prices of the nodes of P are upper bounded by N in view of Eqs. (4) and (5). Next, we observe that there is a price change of at least one unit with each contraction, and since the prices of the nodes of P are upper bounded by N, there can be only a finite number of contractions. Since there can be at most $N-1$ successive extensions without a contraction, the algorithm must terminate. Since, throughout the algorithm, $p(N) = 0$ and the condition $p(i) \leq p(j)+1$ holds for all arcs (i,j), the existence of a forward path starting at a node n_1 and ending at N implies that $p(n_1) < N$

throughout the algorithm. Therefore, if termination occurs via Step 3, there cannot exist a path from n_1 to N. □

Improved Version of the Algorithm. Most of the calculation in the preceding algorithm is needed to determine the nodes $\text{succ}(n_t)$ attaining the minimum in Eq. (6) of Step 1. On the other hand, typically some of these nodes and the corresponding arcs do not change frequently during the algorithm. Thus, it makes sense to save them in a data structure and try to reuse them as much as is possible without affecting the essential properties of the algorithm [maintaining conditions (3)–(5) and precluding the formation of a cycle within the path P]. This leads to a modification of the algorithm, where in addition to the price vector p, we maintain for each node $i \neq N$, a subset of outgoing arcs of i denoted $\text{Cand}(i)$, and called the candidate set of arcs of node i. The set of end nodes of arcs in $\text{Cand}(i)$ which are opposite to i is denoted $\text{Succ}(i)$.

The sets of arcs $\text{Cand}(i)$ together with the set of prices $p(i)$ define a graph, called the admissible graph, whose node set is $\mathcal{N} = \{1, \ldots, N\}$ and arc set is

$$\{(i,j) \mid j \in \text{Succ}(i), p(i) \geq p(j), i = 1, \ldots, N\}.$$

As the sets $\text{Succ}(i)$ and the prices $p(i)$ change in the course of the algorithm, the admissible graph also changes. We require that the initial sets $\text{Cand}(i)$ and prices $p(i)$ are such that the admissible graph is acyclic. This condition is satisfied in particular if we select the sets $\text{Cand}(i)$ to be empty. The algorithm is as follows:

Algorithm A2. Path Construction Algorithm: Second Version. Set $P = (n_1)$, and select p such that Eqs. (3) and (4) hold.

Step 1. Check for Contraction or Extension. Let n_t be the terminal node of P. If there is a node $\bar{j} \in \text{Succ}(n_t)$ such that

$$p(n_t) \geq p(\bar{j}), \tag{10}$$

select such a node $\bar{j}$ and go to Step 2. Otherwise, if the set $N(n_t)$ is empty, set $p(n_t) = N$ and go to Step 3; otherwise, set

$$\text{Succ}(n_t) = \left\{ \bar{j} \;\middle|\; (\bar{j}) = \min_{j \in N(n_t)} p(j) \right\}, \tag{11}$$

$$\text{Cand}(n_t) = \{(n_t, \bar{j}) \in A(n_t) \mid \bar{j} \in \text{Succ}(n_t)\}, \tag{12}$$

and select a node $\bar{j} \in \text{Succ}(n_t)$. Set

$$p(n_t) = p(\bar{j}) + 1. \tag{13}$$

If $n_t = n_1$ and $p(n_1) < N$, or if

$$n_t \neq n_1 \text{ and } p(\text{pred}(n_t)) > p(\bar{j}), \tag{14}$$

go to Step 2; otherwise, go to Step 3.

Step 2. Extend Path. Extend P by node $\bar{j}$ and the corresponding arc of $\text{Cand}(n_t)$. If $\bar{j} = N$, terminate the algorithm; otherwise, go to Step 1.

Step 3. Contract Path. If $P = (n_1)$ and $p(n_1) \geq N$, terminate the algorithm; otherwise, contract P and go to Step 1.

Note that, similar to the first version of the algorithm, each contraction is accompanied by an increase of the price $p(n_t)$, while each extension may or may not be accompanied by an increase of $p(n_t)$. Note also that if the downhill test $p(n_t) \geq p(\bar{j})$ of Eq. (10) were to be replaced by the strictly downhill test $p(n_t) = p(\bar{j}) + 1$, the two versions of the algorithm would have been essentially identical [the sets $\text{Cand}(i)$ would just provide a specific implementation of the successor node selection of Eq. (6)]. However, because of the difference in the test for making an extension to a node of $\text{Succ}(n_t)$, the two versions of the algorithm are not mathematically equivalent. In particular, in the second version, we perform an extension when upon entering Step 1, we have

$$p(n_t) = p(\bar{j}), \qquad \text{for some } \bar{j} \in \text{Succ}(n_t),$$

in which case the last arc of the path P is not strictly downhill following the extension. For this reason it is not obvious than an extension will not create a cycle in P with an attendant breakdown of the algorithm.

It turns out, however, that such a cycle cannot be closed, because it can be proved that throughout the algorithm:

(a) The arcs of P belong to the admissible graph.
(b) The admissible graph remains acyclic.

Both of these properties can be shown by induction. In particular, property (a) is maintained because a contraction that deletes the terminal arc of P does not affect the prices of the end nodes of the other arcs of P. Furthermore, each extension is done along an arc of $\text{Cand}(n_t)$ and, whether the test (10) is passed or $p(n_t)$ is set via Eq. (13), this arc is downhill and its predecessor arc continues to be downhill following the extension. Also, to show that property (b) is maintained, suppose that property (b) holds at the start of Step 1, and consider the two cases where a node $\bar{j} \in \text{Cand}(n_t)$ satisfying the downhill test (10) can be found or cannot be found. In the first case,

the admissible graph remains unchanged. In the second case, the only potentially new arcs of the admissible graphs are the arcs of the set Cand(n_t), after this set is recalculated. However, following the price setting of Eq. (13), all the arcs of Cand(n_t) are strictly downhill, so these arcs cannot be part of a cycle of the admissible graph, all the arcs of which are downhill by definition. Thus the admissible graph remains acyclic following Step 2 or 3, which shows that P remains a simple path at all times. We have the following proposition.

Proposition 2.2. Assume that the initial admissible graph is acyclic. Then:

(a) Throughout the algorithm, the admissible graph remains acyclic.
(b) The flow-price pairs generated by the algorithm satisfy the conditions (3) and (4), the path P is simple, and the arcs of P are downhill.
(c) If there exists a path from n_1 to N, the algorithm terminates via Step 2 with such a path. Otherwise, the algorithm terminates via Step 3.

Proof. Part (a) was shown above and the remaining parts are proved similarly to the corresponding parts of Proposition 2.1. □

3. Auction/Max-Flow Algorithm

We now consider the max-flow problem as described in Section 1. We introduce some additional terminology.

Given a capacity feasible flow vector x, for each node i, we introduce the set of eligible arcs of i, given by

$$A(i, x) = \{(i, j) \mid x_{ij} < c_{ij}\} \cup \{(j, i) \mid 0 < x_{ji}\}, \tag{15}$$

and the corresponding set of eligible neighbors of i, given by

$$N(i, x) = \{j \mid (i, j) \in A(i, x) \text{ or } (j, i) \in A(i, x)\}. \tag{16}$$

The reduced graph is the graph with node set $\mathcal{N}$ which contains an arc (i, j) if and only if j is an eligible neighbor of i. Thus, eligible arcs of a node i in the original graph correspond to outgoing arcs from i in the reduced graph. For a given capacity feasible x, a path P in the original graph is said to be unblocked if it corresponds to a forward path of the reduced graph, that is, if $x_{ij} < c_{ij}$ for all forward arcs $(i, j) \in P^+$ and $0 < x_{ij}$ for all backward arcs $(i, j) \in P^-$. An unblocked path is said to be augmenting if its start node has

positive surplus and its terminal node is the sink N. If P is an augmenting path, an augmentation is an operation that increases the flow of all arcs $(i,j)\in P^+$ and decreases the flow of all arcs $(i,j)\in P^-$ by a common increment $\delta>0$.

Following standard terminology, a cut is a partition $(\mathcal{N}^+, \mathcal{N}^-)$ of the set of nodes $\mathcal{N}$ into two subsets $\mathcal{N}^+$ and $\mathcal{N}^-$ with $1\in\mathcal{N}^+$ and $N\in\mathcal{N}^-$. The capacity of this cut is the sum of the capacities of all arcs (i,j) with $i\in\mathcal{N}^+$ and $j\in\mathcal{N}^-$. The max flow-min cut theorem states that the maximum flow is equal to the minimal cut capacity. For a given flow vector x, a cut $(\mathcal{N}^+, \mathcal{N}^-)$ is said to be saturated if $x_{ij}=c_{ij}$ for all arcs (i,j) with $i\in\mathcal{N}^+$ and $j\in\mathcal{N}^-$, and $x_{ij}=0$ for all arcs (i,j) with $i\in\mathcal{N}^-$ and $j\in\mathcal{N}^+$. The algorithm of this section terminates with a capacity feasible flow vector x and a cut $(\mathcal{N}^+, \mathcal{N}^-)$ that is saturated and is such that the surpluses g_i, given by Eq. (1), satisfy

$$g_1\le 0, \tag{17a}$$

$$g_i\ge 0, \qquad \forall i\ne 1, \tag{17b}$$

$$g_i=0, \qquad \forall i\in\mathcal{N}^-,\ i\ne N. \tag{17c}$$

It is well known that such a cut is a minimum cut, and we will show how it can be used together with x to obtain a maximum flow; see the remarks following the proof of Proposition 3, which also prove that the cut obtained upon termination is minimum.

A capacity feasible flow vector x together with a price vector

$$p=\{p(i)\mid i\in\mathcal{N}\}$$

are said to be a valid pair if

$$p(i)\le p(j)+1, \qquad \forall j \text{ that are eligible neighbors of } i. \tag{18}$$

Our algorithm starts with and maintains a valid flow-price pair (x,p) such that

$$g_1\le 0, \qquad g_i\ge 0, \forall i\ne 1, \tag{19}$$

$$p(1)=N, \qquad p(N)=0, \qquad p(i)\ge 0, \forall i\ne 1, N. \tag{20}$$

A possible initial choice is the flow vector x given by

$$x_{ij}=\begin{cases} c_{ij}, & \text{if } i=1,\\ 0, & \text{if } i\ne 1, \end{cases} \tag{21a}$$

together with the price vector p given by

$$p(i)=\begin{cases}N, & \text{if } i=1,\\ \text{length of a shortest unblocked path from } i \text{ to } N, & \text{if } i\neq 1,\end{cases} \tag{21b}$$

which can be obtained by a breadth-first search starting from N. If there is no forward path of the original graph from i to N, the above length is taken to be equal to N.

Our algorithm maintains a flow-price pair (x, p) satisfying the conditions (18)–(20), performs a sequence of iterations, and terminates with a minimum cut. At the start of each iteration, a node n_1 with

$$n_1\neq N, \qquad p(n_1)<N, \qquad g_{n_1}>0$$

is selected. The iteration tries to construct an augmenting path starting at n_1 by using the second path construction algorithm of the preceding section, applied to the reduced graph and using the price vector p. If an augmenting path is found, the iteration concludes with a corresponding augmentation. If an augmenting path cannot be found, the path construction algorithm terminates with $p(n_1)\geq N$, so that node n_1 will not be chosen as the starting node at any subsequent iteration. Consistently with the second path construction algorithm of Section 2, we maintain, for each node i, a set of incident arcs of i denoted Cand(i). The set Cand(i) is empty for $i=1$, $i=N$, and all i with $p(i)=N$. The set of end nodes of Cand(i) which are opposite to i is denoted Succ(i).

We require that, initially, we have

$$p(i)=p(j)+1, \qquad \text{if } j\in \text{Succ}(i),$$

which will be true if all sets Cand(i) are empty or for all i we have

$$\begin{aligned}\text{Cand}(i)=&\{(i,j)\in A(i,x)\mid p(i)=p(j)+1\}\\ &\cup\{(j,i)\in A(i,x)\mid p(i)=p(j)+1\},\end{aligned} \tag{22}$$

where (x, p) is the initial flow-price pair. If the shortest path initialization of Eqs. (21a) and (21b) is used, then Cand(i) as given by Eq. (22) is the set of arcs outgoing from i in a shortest augmenting path from i. The typical iteration is as follows.

Algorithm A3. Auction/Max-Flow Typical Iteration. Select a node n_1 with $n_1\neq N$, $p(n_1)<N$, and $g_{n_1}>0$; if no such node exists, the algorithm terminates. Set $P=(n_1)$.

Step 1. Check for Contraction or Extension. Let n_t be the terminal node of P. If there is a node $\bar{j} \in \text{Succ}(n_t) \cap N(n_t, x)$ such that

$$p(n_t) \geq p(\bar{j}), \tag{23}$$

select such a node $\bar{j}$ and go to Step 2. Otherwise, if the set $N(n_t, x)$ is empty, set $p(n_t) = N$ and go to Step 3; otherwise, set

$$\text{Succ}(n_t) = \left\{ \bar{j} \,\middle|\, p(\bar{j}) = \min_{j \in N(n_t, x)} p(j) \right\}, \tag{24}$$

$$\text{Cand}(n_t) = \{(n_t, j) \in A(n_t, x) \mid j \in \text{Succ}(n_t)\} \cup \{(j, n_t) \in A(n_t, x) \mid j \in \text{Succ}(n_t)\}, \tag{25}$$

and select a node $\bar{j} \in \text{Succ}(n_t)$. Set

$$p(n_t) = p(\bar{j}) + 1. \tag{26}$$

If $n_t = n_1$ and $p(n_1) < N$, or if

$n_t \neq n_1$ and $p(\text{pred}(n_t)) > p(\bar{j})$,

go to Step 2; otherwise, go to Step 3.

Step 2. Extend Path. Extend P by the node $\bar{j}$ and the corresponding arc of $\text{Cand}(n_t)$. If $\bar{j}$ is the sink N, go to Step 4; otherwise, go to Step 1.

Step 3. Contract Path. If $P = (n_1)$ and $p(n_1) \geq N$, terminate the iteration; otherwise, contract P and go to Step 1.

Step 4. Augmentation. Perform an augmentation along P with flow increment

$$\delta = \min\{g_{n_1}, \{c_{ij} - x_{ij} \mid (i,j) \in P^+\}, \{x_{ij} \mid (i,j) \in P^-\}\}, \tag{27}$$

and terminate the iteration.

Note that, except for the at most $N-2$ contractions in which $p(n_t)$ is set to N, all contractions involve an increase of the price $p(n_t)$ and a recalculation of the set $\text{Succ}(n_t)$. Extensions can occur either through a discovery of a node $\bar{j} \in \text{Succ}(n_t) \cap N(n_t, x)$ such that $p(n_t) \geq p(\bar{j})$ or through a recalculation of the set $\text{Succ}(n_t)$, in which case an increase of $p(n_t)$ may or may not occur.

We assume that the search through the set $\text{Succ}(n_t) \cap N(n_t, x)$ for a node $\bar{j}$ such that $p(n_t) \geq p(\bar{j})$ is organized as follows. When a set $\text{Cand}(i)$ is initially calculated, via, for example, Eq. (22), or is recalculated via Eq. (25),

it is organized as a queue, which allows the deletion of its top element with $O(1)$ work. Each iteration is started by retrieving sequentially arcs from the top of $\text{Cand}(n_t)$ and checking to see if these arcs are eligible and their endnode $\bar{j}$ opposite to n_t satisfies $p(n_t) \geq p(\bar{j})$. Each arc not passing these tests is deleted from $\text{Cand}(n_t)$, and the checking is stopped when either a node $\bar{j}$ with the required properties is found or the set $\text{Cand}(n_t)$ becomes empty. To simplify the following complexity accounting, the work for checking and deleting the arcs of $\text{Cand}(n_t)$ is lumped into the work for calculating $\text{Cand}(n_t)$. With this convention, the work involved in an extension for which we recalculate the set $\text{Cand}(n_t)$ via Eq. (25) is proportional to the degree of n_t, while the work involved in an extension, where after checking and possibly deleting enough arcs of $\text{Cand}(n_t)$ we find an eligible neighbor node $\bar{j}$ that passes the test $p(n_t) \geq p(\bar{j})$, is $O(1)$. Similarly, the work involved in a contraction is proportional to the degree of n_t.

The next proposition establishes the basic properties of the algorithm.

Proposition 3.1. The following hold for the max-flow algorithm of this section:

(a) Each iteration of the max-flow algorithm up to the discovery of the corresponding augmenting path consists of an application of the second path construction algorithm of the preceding section to the reduced graph, with the start node of the path being the chosen node n_1 for this iteration, and the end node of the path being N.

(b) The algorithm terminates and, upon termination, there is a saturated cut separating the sink from all nodes with nonzero surplus, which is a minimum capacity cut.

(c) The running time of the algorithm is $O(N^2A)$.

Proof.

(a) By comparing the descriptions of the second path construction algorithm and the iteration of the max-flow algorithm, we see that the condition (3) that is maintained by the path construction algorithm is equivalent to the condition (18) for the pair (x, p) to be valid, the price change (13) corresponds to the price change (26), and the downhill test (10) for an extension corresponds to the downhill test (23). Let us define the admissible graph of the max-flow algorithm as the graph whose node set is $\mathcal{N} = \{1, \ldots, N\}$ and arc set is

$$\{(i, j) \mid j \in \text{Succ}(i) \cap N(i, x), p(i) \geq p(j), i = 1, \ldots, N\}.$$

Then the sets Succ(i) and the admissible graph of the path construction algorithm correspond to the sets Succ(i) $\cap N(i, x)$ and the admissible graph in the max-flow algorithm, respectively.

Based on the preceding associations, it is seen that, if at the start of an iteration of the max-flow algorithm the admissible graph is acyclic, then the iteration up to the discovery of an augmenting path is equivalent to the application of the path construction algorithm to the reduced graph. Thus, to prove the result, we must show that the admissible graph of the max-flow algorithm remains acyclic throughout the algorithm.

To this end, we note that, in view of the initial restriction $p(i)=p(j)+1$ for all $j\in$Succ(i), the admissible graph is acyclic at the start of the algorithm. Furthermore, if the admissible graph is acyclic at the start of an iteration, the same is true during the iteration up to the discovery of the augmenting path, since the path construction algorithm maintains the acyclicity of the admissible graph. We claim that an augmentation does not add any new arcs to the admissible graph, and thus maintains its acyclicity. Indeed, suppose that an augmentation occurs along the path $(i_1, i_2, \ldots, i_k, N)$, and that one of the arcs (i_m, i_{m-1}), $m=2, \ldots, k$, is added to the reduced graph and to the admissible graph as a result of the augmentation. Then, we must have

$$p(i_m)\geq p(i_{m-1}), \qquad i_{m-1}\in \text{Succ}(i_m),$$

by the definition of the admissible graph, and also

$$p(i_{m-1})\geq p(i_m), \qquad i_m\in \text{Succ}(i_{m-1}),$$

since the arc (i_{m-1}, i_m) belongs to the augmenting path, so that $p(i_{m-1})=p(i_m)$. This implies that $p(i_{m-1})$ and $p(i_m)$ have been increased at least once since the start of the algorithm [since we have $p(i)=p(j)+1$ for all $j\in$Succ(i) at the start of the algorithm and also following each recalculation of the set Succ(i)]. Furthermore, the conditions $p(i_{m-1})<p(i_m)+1$ and $i_m\in$Succ(i_{m-1}) imply that the last increase of $p(i_m)$ occurred after the last recalculation of Succ(i_{m-1}) [since following a recalculation of Succ(i) at a node i, we have $p(i)=p(j)+1$ for all $j\in$Succ(i)]. Therefore the last increase of $p(i_m)$ occurred after the last increase of $p(i_{m-1})$ [since each increase of $p(i)$ involves a recalculation of Succ(i)]. Similarly, the conditions $p(i_m)<p(i_{m-1})+1$ and $i_{m-1}\in$Succ(i_m) imply that the opposite is true. We thus reach a contradiction.

(b) From part (a) and Proposition 2.2, it follows that each iteration terminates. At the end of an iteration, either we have $p(n_1)\geq N$, indicating that there is no augmenting path starting at n_1, or we have an augmentation. In the former case, the number of nodes i with $p(i)\geq N$ increases strictly, so there can be at most $N-2$ iterations of this type. To show that the number of augmentations is finite, we first note that there are at most N

price increases per node, since prices take nonnegative integer values, and once the price of a node exceeds $N-1$, it increases no further. We next observe that each augmentation either exhausts the surplus of n_1, or saturates at least one arc (that is, it drives the flow of the arc to zero or its upper bound). When an arc with end nodes i and j is saturated in the direction from i to j, there are two possibilities,

$$\text{Case 1}: p(i)=p(j)+1, \qquad \text{Case 2}: p(i)=p(j).$$

In Case 2, since $j\in\text{Succ}(i)$, we cannot have $i\in\text{Succ}(j)$, since this would violate the acyclicity of the admissible graph. In either Case 1 or Case 2, we see that one of the at most N increases of $p(j)$ must occur before this arc can become unsaturated and then saturated again in the direction from i to j. Thus, the number of arc saturations is $O(N)$ per arc, and the total number of arc saturations is $O(NA)$, leading to an $O(NA)$ bound in the number of iterations and the number of augmentations.

We thus see that the algorithm terminates, and since augmentations preserve the condition $g_i\geq 0$ for all $i\neq 1$, upon termination, we must have $g_i\geq 0$ for all $i\neq 1$, $p(1)=N$, $p(i)\geq N$ for all $i\neq N$ with $g_i>0$, and $p(N)=0$. It follows that there can be no augmenting path starting at node 1 or at a node i with $g_i>0$, implying that there is a saturated cut $(\mathcal{N}^+, \mathcal{N}^-)$ such that $1\in\mathcal{N}^+$, $N\in\mathcal{N}^-$, $g_i\geq 0$ for all $i\neq 1$, and $g_i=0$ for all $i\neq N$ with $i\in\mathcal{N}^-$. As discussed earlier, this is a minimum cut.

(c) We first note that, as shown in the proof of part (b):

(i) There are at most N price increases per node.
(ii) There are at most $O(NA)$ iterations and at most $O(NA)$ augmentations.

In view of (i) above, there can be at most N contractions and extensions that involve a price increase at each node, and the work for each is proportional to the degree of n_t. Thus, the work for these contractions and extensions is $O(NA)$. Also, since each augmentation involves a flow change for each of at most $N-1$ arcs, the work for augmentations is $O(N^2A)$.

There remains to bound the work for extensions that do not involve a price increase. We argue by contradiction that each such extension does not involve the recalculation of the set $\text{Succ}(n_t)$, that is, either it involves the first calculation of $\text{Succ}(n_t)$ or the downhill test (23) is failed for all $\bar{j}\in\text{Succ}(n_t)\cap N(n_t, x)$. Indeed, suppose that the set $\text{Succ}(n_t)$ is recalculated via Eq. (24) and we find that $p(n_t)=p(j)+1$ for all $j\in\text{Succ}(n_t)$, so that an extension is performed without an increase of $p(n_t)$. Then, every $j\in\text{Succ}(n_t)$ must have been an eligible neighbor of n_t and its price must have remained unchanged continuously since the preceding time $\text{Succ}(n_t)$ was calculated [and $p(n_t)$ was set to $p(j)+1$]. But this is a contradiction, since in order for

$\text{Succ}(n_t)$ to be recalculated, all nodes j in the set $\text{Succ}(n_t) \cap N(n_t, x)$ must satisfy $p(n_t) < p(j)$. Thus, if an extension at n_t does not involve a price increase, it also does not involve a recalculation of $\text{Succ}(n_t)$; therefore, using the accounting method described in the paragraph preceding Proposition 3.1, it requires only $O(1)$ work, unless it involves the calculation of $\text{Succ}(n_t)$ for the first time. Now, the total number of extensions is $O(N^2A)$, because in each iteration the number of extensions exceeds the number of contractions by at most $N-1$, the total number of contractions in the entire algorithm is $O(N^2)$, while the total number of iterations is $O(NA)$. Thus, the total work for extensions that do not involve a price increase is $O(N^2A)$. □

Given the cut $(\mathcal{N}^+, N^-)$ and the flow vector x obtained upon termination of the algorithm, we can obtain a maximum flow by applying the same algorithm to a certain feasibility problem, that aims to return to the source the excess flow that has entered the graph from the source and has accumulated at the other nodes of $\mathcal{N}^+$. In particular, we delete all nodes in $\mathcal{N}^-$ and all arcs with at least one endnode in $\mathcal{N}^-$, and for each node $i \neq 1$ with $i \in \mathcal{N}^+$ and

$$\sum_{\{(i,j) \mid j \in \mathcal{N}^-\}} c_{ij} > 0, \tag{28}$$

we introduce an arc $(i, 1)$ with flow and capacity

$$x_{i1} = c_{i1} = \sum_{\{(i,j) \mid j \in \mathcal{N}^-\}} c_{ij}; \tag{29}$$

if the arc $(i, 1)$ already exists, we just change its capacity and flow to the above value. In the resulting graph, call it $\mathcal{G}$, we pose the problem of finding a flow vector $\bar{x}$ such that the corresponding surpluses are all zero. It can be seen that the surpluses corresponding to the flow vector x restricted to $\mathcal{G}$ are equal to the nonnegative surpluses g_i obtained upon termination for all $i \neq 1$. We can thus apply the max-flow algorithm of this section starting with this flow vector, and the prices

$$p(i) = \begin{cases} 0, & \text{if } i = 1, \\ \text{length of a shortest unblocked path from } i \text{ to } 1, & \text{if } i \neq 1, \end{cases}$$

which together with x form a valid pair for the graph $\mathcal{G}$. It can be shown then that each iteration of the algorithm will terminate with an augmentation from some node i with $g_i > 0$ to the source 1. [Given any capacity feasible flow vector in a graph with arc capacities, and a node i with positive surplus, there is always an augmenting path starting at i and ending at some node with negative surplus; this follows from the conformal realization theorem

(see, e.g., Ref. 5, p. 7). Here node 1 is the only node with negative surplus.] Thus, the algorithm will terminate when the surpluses of all the nodes $i \neq 1$ will be reduced to 0, while upon termination the flows of the arcs $(i, 1)$ will still be equal to their initial values given by Eq. (29), since these arcs cannot participate in an augmenting path. If $\bar{x}_{ij}$ is the final flow of each arc (i, j) of $\mathcal{G}$, it can be seen, using also the fact $g_i = 0$ for all $i \in \mathcal{N}^-$ with $i \neq N$, that the flow vector x^* defined for each arc $(i, j) \in \mathcal{A}$ by

$$x_{ij}^* = \begin{cases} \bar{x}_{ij}, & \text{if } i \in \mathcal{N}^+, j \in \mathcal{N}^+, \\ x_{ij}, & \text{otherwise,} \end{cases}$$

will have surpluses g_i^* satisfying $g_i^* = 0$ for all $i \neq 1, N$, $g_1^* < 0$, $g_N^* > 0$, while saturating the cut $(\mathcal{N}^+, \mathcal{N}^-)$. Thus, by the max flow-min cut theorem, x^* must be a maximum flow and $(\mathcal{N}^+, \mathcal{N}^-)$ must be a minimum cut.

Note, from the proof of Proposition 3.1, that the complexity bottleneck is the $O(N^2A)$ bound for augmentations and for extensions that do not involve a price increase. Our computational experience, however, indicates that the $O(NA)$ work for price increases is at least as much of a bottleneck. This is similar to preflow-push methods where the $O(NA)$ work for price increases usually dominates the computation, even though the worst case complexity bound is worse than $O(NA)$. It thus appears that the practical computation bottlenecks are comparable for preflow-push methods and our method.

We finally note two variants of the max-flow algorithm. In the first variant, we use the first version of the path construction algorithm, given in Section 2, in place of the second version. The statement of the typical iteration of this algorithm is identical with the one given above, except that the downhill test $p(n_t) \geq p(\bar{j})$ of Eq. (23) is replaced by the strictly downhill $p(n_t) = p(\bar{j}) + 1$. Proposition 3.1 can also be proved for this variant of the algorithm using a similar (in fact, simpler) proof.

In the second variant of the max-flow algorithm, instead of maintaining the entire set Cand(i), we maintain just one arc of Cand(i). The iteration of the algorithm is modified so that, if the unique arc of Cand(n_t) passes the downhill test of Eq. (23), it is used as earlier. Otherwise, assuming that $N(n_t, x)$ is nonempty, the set Succ(n_t) is computed and a single arc of Cand(n_t) is retained. This variant can be shown to terminate with a minimum cut as stated in Proposition 3.1. Its complexity analysis is similar to the one given in the proof of Proposition 3.1(c), except that the work for extensions that do not involve a price increase can be estimated as $O(NA^2)$ rather than $O(N^2A)$, raising the complexity bound to $O(NA^2)$. However, when combined with the second best data structure given in the next section, this second variant of the max-flow algorithm proved the most effective in our computational results.

4. Efficient Implementation

In this section, we describe a number of variations of the auction/max-flow algorithm of the preceding section, which have been found empirically to improve performance.

Tests for a Saturated Cut. It has been observed that, for some problems (particularly those involving a sparse graph), our method can create a saturated cut very quickly and may then spend a great deal of additional time to raise to the level N the prices of the nodes that are left with positive surplus. This characteristic is shared with preflow-push methods. Computational studies (Refs. 16, 17, 19, 20) of preflow-push methods have shown that it is extremely important to use a procedure that detects early the presence of a saturated cut. Several schemes have been suggested in the literature.

One possibility is to test periodically for a saturated cut by an $O(A)$ breadth-first search from the sink, which tries to find the set $\mathcal{S}$ of nodes from which there is an unblocked path to the sink. If all nodes in $\mathcal{S}$ have zero surplus, then $\mathcal{S}$ defines a minimum cut. Note that, once a node of $\mathcal{S}$ with positive surplus is found, the breadth-first search can be terminated. However, in an alternative version of this scheme, one can also perform global repricing, whereby all the nodes in $\mathcal{S}$ are obtained, and their prices are recalculated and are set to their shortest distances from the sink. Furthermore, all the nodes not in $\mathcal{S}$ can effectively be purged from the computation by setting their price equal to N. While global repricing can be costly, it is known to be beneficial for several problem types (Refs. 17, 19, 20). It is important to use an appropriate heuristic scheme that ensures that global repricing is not too frequent, in view of the associated overhead. In practice, repeating the test after a number of contractions, which is of the order of N, seems to work well.

Another possibility, suggested in the context of preflow-push methods in Ref. 16, is to maintain in a suitable data structure, for each integer k in the range $[1, N-1]$, the number of nodes $m(k)$ whose price is equal to k. If for some k we have $m(k)=0$ (this is called a gap at price k), then there is a saturated cut separating all nodes with price greater than k from all nodes whose price is less than k.. All the nodes with price greater than k can be purged effectively from the computation by setting their price equal to N. Furthermore, if all nodes with price less than k have zero surplus, the separating saturated cut is a minimum cut. In our experiments, we have found this second procedure in conjunction with the highest price selection rule to be more effective than the first. Note an advantage of both of these procedures:

they can purge from the computation a significant number of nodes before finding a minimum cut.

Method for Selecting the Starting Node of the Path. Our algorithm leaves unspecified the choice of the positive surplus node used as the starting node of the path P. One possibility is to select a node with the highest price among all positive surplus nodes i with $p(i) < N$. Each time the path P degenerates to its start node, following a contraction, it is possible to make a new start node selection based on the highest price criterion without affecting the termination properties of the algorithm.

An alternative is to maintain all nodes i with positive surplus and $p(i) < N$ in a FIFO queue, and use as starting node the first node in the queue. Note that the preflow-push method that uses a highest price scheme is superior to the method that uses a FIFO scheme in terms of worst-case complexity [$O(N^2A^{1/2})$ versus $O(N^3)$].

Greedy Augmentations. Once an augmenting path is constructed, instead of pushing the same amount of flow along each arc of the path, it is possible to push along each arc (i,j) the maximum possible amount of flow, that is, $\max\{g_i, c_{ij}-x_{ij}\}$, if (i,j) is a forward arc of the path, or $\max\{g_j, x_{ij}\}$, if (i,j) is a backward arc of the path. We call this a greedy augmentation. For an example where such augmentations are helpful, see Fig. 1.

There is a possible weakness of our algorithm that cannot be corrected via greedy augmentations. This arises when many augmentations involve

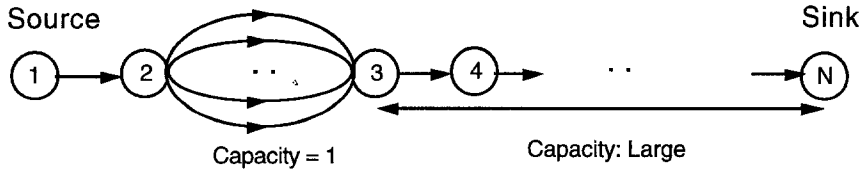

Fig. 1. An example where greedy augmentations are helpful. When the augmentations are done as in the preceding section [cf. Eq. (27)], flow moves in single units along the long path from node 2 to node N, and the number of arc flow changes is $m(N-2)$, where m is the number of arcs joining nodes $N-1$ and N. If greedy augmentations are used instead, the first augmentation moves a large amount of flow to node $N-1$, and all subsequent augmentations involve a single-arc path from $N-1$ to N. The number of arc flow changes is $m+N-3$.

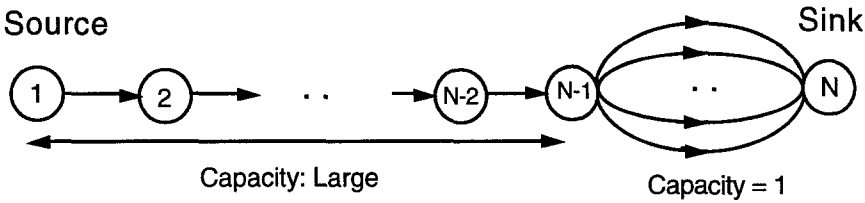

Fig. 2. A potential weakness of the auction/max-flow algorithm that cannot be corrected through greedy augmentations. In this example, flow moves in single units along the long paths from node 2 to node N.

small increments along long paths. For an example, see Fig. 2. In the implementation used in our tests, we have employed a heuristic procedure that identifies situations of this type, and appropriately compensates for it by occasionally moving flow along shorter portions of very long paths. However, even without this heuristic procedure, our method outperforms substantially preflow-push methods in the experiments reported in the next section.

Using a Second Best Candidate. Consider the variant of the algorithm, where only one node of the set $\mathrm{Succ}(i)$, call it $j_1(i)$, is maintained for each i, together with a corresponding arc of $\mathrm{Cand}(i)$. Suppose that, for the terminal node n_t of the current path P, we have available a lower bound $\beta(n_t)$ on the prices of all the nodes in $N(n_t, x)$, except for the price of node $j_1(n_t)$. Suppose also that, in Step 1, the downhill test $p(n_t) \geq p(j_1(n_t))$ of Ineq. (23) for an extension is failed. Then, we can check to see whether we have

$$p(j_1(n_t)) \leq \beta(n_t),$$

and if this is so, we know that $p(j_1(n_t))$ is still less or equal to the prices of all nodes in $N(x, n_t)$, thereby making the computation of this minimum as per Eq. (24) unnecessary. A lower bound of this type can be obtained by calculating, together with $j_1(n_t)$, the second best node in $N(n_t, x)$ that is, a node $j_2(n_t)$ given by

$$j_2(n_t) = \arg \min_{j \in N(n_t, x), j \neq j_1(n_t)} p(j).$$

Then, as long as $j_1(n_t)$ remains unchanged and no new node is added to $N(n_t, x)$, we can use

$$\beta(n_t) = p(j_2(n_t))$$

as a suitable lower bound [if a new node is added to $N(n_t, x)$ due to an augmentation, we must suitably modify $\beta(n_t)$ and $j_2(n_t)$]. This idea can be further strengthened by checking to see if $j_2(n_t)$ still belongs to $N(n_t, x)$ and

whether its price is still $\beta(n_t)$, in the case where the test $p(j_1(n_t)) \leq p(j_2(n_t))$ is failed. If this is so, we can set $j_1(n_t)$ to $j_2(n_t)$, thereby obviating again the calculation of the minimum in Eq. (24).

The idea of using a second best candidate arc and node is known to be very effective in auction algorithms for the assignment problem (Ref. 5, p. 176 and Ref. 23) and the shortest path problem (Refs. 21, 24). It similarly improves the performance of our max-flow algorithm.

5. Computational Results

To test the ideas of this paper, we have developed a FORTRAN code, called AUCTION-MF, which is based on the variant of the algorithm that maintains just a single element of each set Cand(i). This code is available to use for research purposes from the author (bertsekas@lids.mit.edu). The code uses some of the implementation ideas of Section 4 as follows:

(a) Greedy augmentations.
(b) Choice of the highest price node with positive surplus as the starting node, and use of a gap scheme for saturated cut detection.
(c) The second best candidate data structure.

These implementation ideas gave the best results for the problems tested.

We have experimented with several other versions of the code, which differ in the way they select the starting node of the path and in the way they detect the presence of a saturated cut. For example, we have tested a code that, instead of (b) above, maintains the positive surplus nodes in a cyclic queue, chooses the top node of the queue as the starting node, and uses periodic breadth-first search for saturated cut detection and global repricing. This version performed quite well relative to preflow-push methods, but was uniformly slower than AUCTION-MF.

Random Problem Generators. We have used for experimentation test problems obtained with a variety of standard random problem generators. Max-flow problems generated by several types of random problem generators tend to be easy and can be solved very quickly by state-of-the-art codes soon after initialization. Since all the codes we tested use very similar initialization, based on breadth-first search [cf. Eqs. (21), (22)], we have focused our experimentation on problems that are quite difficult and require considerable computation beyond initialization. These problems are typically characterized by large differences between the initial and the final price vectors, as well as a large number of price changes.

RMFGEN. This code generates three-dimensional grid graphs, as described in Ref. 25. The problem is specified by two parameters a and b, called the side and the height, respectively. The grid has b frames numbered consecutively, each consisting of a two-dimensional grid of a^2 nodes. Node 1 of frame 1 is the source, and node a^2 of frame b is the sink. Within a frame, each pair of neighbor nodes is connected with an arc of capacity 10^3a^2. Between two successive frames, say k and $k+1$, there are $2a^2$ arcs; a^2 arcs that start at nodes of frame k and end at nodes of frame $k+1$, and another a^2 arcs that start at nodes of frame $k+1$ and end at nodes of frame k. These arcs have capacity randomly chosen from the integer range $[1, 10^3]$. Furthermore, the end nodes of these arcs are determined by a random permutation rule; that is, for each node of a frame, there must be exactly one arc incoming from, and exactly one arc outgoing to, each of the neighboring frames. There are several possible permutation rules. The one that we used for our experiments works as follows. The nodes of each frame are sequentially numbered from 1 to a^2; that is, the first row consists of nodes 1 to a, the second row consists of nodes $a+1$ to a^2, etc. For each ordered pair of neighboring frames, an integer r is randomly chosen from the range $[1, a^2]$. Then, for each node i of the first frame, an arc is created that starts at i and ends at node $r+i$ modulo a^2 of the second frame. This random permutation rule results apparently in more challenging problems than those obtained using other permutation rules, such as a^2 successive random interchanges of node pairs.

GRID-SQ. This code generates a two-dimensional square grid problem. The source is connected to all nodes of the bottom row of the grid with arcs of very large capacity. All nodes of the top row of the grid are connected to the sink with arcs of very large capacity. Also, each node of the grid is connected to all its immediate neighbors with an arc of capacity randomly chosen from the integer range $[1, 10^6]$.

NETGEN. This is a standard generator described in Ref. 26, which generates random graphs with given number of nodes and arcs, and with capacities chosen from a given range. A tree of arcs of high capacity, whose value is specified by the user, connects all nodes.

Codes Used for Comparison. Extensive computational studies (e.g., Refs. 16, 17, 19, 20) have established that preflow-push algorithms are the fastest of the presently available max-flow methods. We have accordingly compared our auction code with two state-of-the-art preflow-push codes.

These are:

PFP-AO. This is a FORTRAN code due to Ahuja and Orlin. It is an efficient implementation of the preflow-push method with the highest price selection rule and the gap scheme for saturated cut detection.

PFP-DM. This is a FORTRAN code due to Derigs and Meier (Ref. 16). It is a preflow-push method that is similar to PFP-AO in that it also uses the highest price selection rule and the gap scheme for saturated cut detection. However, the implementations of PFP-AO and PFP-DM are somewhat different.

AUCTION-MF, PFP-AO, and PFP-DM were tested under identical conditions on two machines:

(a) A Macintosh IIci with 32 Megabytes of memory using the Absoft FORTRAN compiler.
(b) A DECStation 5025 with 128 Megabytes of memory using the FORTRAN compiler under UNIX.

We have also performed some experimentation on the NeXTStation 68040 running UNIX with the code of Anderson and Setubal (Ref. 19). This is an efficient implementation in C of the preflow-push method that uses a FIFO node selection rule, periodic breadth-first tests for a saturated cut, and global repricing. The results of this experimentation were consistent with the results given here for the other two preflow-push codes, and can be found in an earlier report (Ref. 1). However, in our experience, comparisons between C and FORTRAN codes tend to be highly unreliable because of the compiler differences, and for this reason we will not present these results.[4]

Summary of Results. We have found that the auction code outperforms substantially the preflow-push codes for all problem classes tested. The closest competitor depends on the problem class. Our algorithm is faster than the closest competitor by at least two to three times for all problem classes. Significantly, it consistently outperforms (sometimes by an order of magnitude) the Ahuja and Orlin code and the Derigs and Meier code, which use a similar node selection rule (highest price) and the same termination scheme (gap detection).

The comparison of the computation times is corroborated by other statistics, which are independent of the computer and the compiler used. In

[4]A line-by-line translation into C of RELAXTII, the FORTRAN code given in Ref. 27 that implements the relaxation method for minimum cost flow, runs consistently about three times faster than the FORTRAN version on a UNIX workstation (Ref. 28).

particular, we have recorded for each code the average number of flow changes per arc and the average number of price changes per node. Generally, the auction algorithm performs substantially (and often dramatically) fewer price and arc flow changes relative to the preflow-push algorithms. Note, however, that the ratios between the number of flow and price changes do not faithfully correspond to the ratios of run times because the different codes involve different data structures and varying amounts of overhead.

The results are given in four tables. In these tables, an asterisk indicates that the corresponding problem was not run due to limited memory of the corresponding machine. The top entries in each box give the running time in seconds on a MacIIci and the running time in seconds on a DECStation 5025. The bottom entries in each box give the average number of flow changes per arc and the average number of price changes per node.

Tables 1 and 2 compare AUCTION-MF, PFP-AO, and PFP-DM on RMFGEN problems.

Table 3 compares AUCTION-MF, PFP-AO, and PFO-DM on GRID-SQ problems.

Table 4 compares AUCTION-MF, PFP-AO, and PFP-DM on NETGEN problems.

Table 1. Experiments with constant side RMFGEN problems. Each entry corresponds to an average over 5 problems.

Side	Height	AUCTION-MF	PFP-AO	PFP-DM
15	40	12.57/2.659 1.375/6.455	78.09/13.04 7.292/29.40	68.27/9.720 6.340/27.46
15	80	19.49/4.286 1.119/4.303	199.5/36.22 8.944/39.35	179.7/25.55 7.914/36.62
15	120	26.18/5.815 1.023/3.469	283.7/49.67 10.96/50.83	262.1/36.88 7.641/35.62
15	160	33.55/7.623 0.993/3.280	435.4/77.87 9.570/44.20	369.8/52.18 7.998/37.82
15	200	41.24/9.442 0.985/3.198	493.4/95.36 11.00/52.04	378.3/53.34 6.550/30.85
15	240	45.89/10.61 0.934/2.7480	611.1/109.8 9.942/46.84	483.5/68.15 6.920/33.00
15	280	*/12.27 0.928/2.679	*/161.2 11.66/55.85	*/85.82 7.427/35.46
15	320	*/13.96 0.912/2.552	*/162.9 10.57/60.67	*/87.54 6.650/31.70
15	360	*/15.28 0.890/2.411	*/177.1 10.07/48.19	*/94.98 6.343/30.34

Table 2. Experiments with constant height RMFGEN problems. Each entry corresponds to an average over 5 problems.

Side	Height	AUCTION-MF	PFP-AO	PFP-DM
10	40	3.837/0.694 1.050/3.693	16.15/2.227 4.383/16.24	14.94/2.335 3.505/13.41
15	40	12.57/2.659 1.375/6.455	78.09/13.04 7.292/29.40	68.27/9.720 6.340/27.46
20	40	30.38/6.758 1.828/9.250	173.7/30.92 10.10/42.91	190.2/28.02 9.617/42.86
25	40	53.07/12.44 1.971/10.59	340.7/61.92 12.04/52.80	325.7/49.66 10.21/46.83
30	40	117.1/26.35 3.117/16.89	667.4/125.8 15.00/69.76	693.9/106.8 14.37/69.70
35	40	162.3/35.88 2.977/16.29	989.4/184.8 15.94/73.56	928.2/146.5 14.20/67.99
40	40	218.8/50.18 3.027/17.61	1526./280.8 18.24/85.55	1377./214.0 15.66/75.72

Table 3. Experiments with GRID-SQ problems. Each entry corresponds to an average over 5 problems.

Side	Height	AUCTION-MF	PFP-AO	PFP-DM
100	100	12.78/2.111 2.095/7.421	26.79/3.438 5.289/13.91	27.94/4.109 4.222/11.06
150	150	31.58/5.670 2.210/8.313	68.66/8.633 5.493/14.90	72.80/10.33 4.547/12.19
200	200	79.75/14.98 2.930/12.65	214.6/31.07 9.246/26.27	223.1/30.59 7.928/22.66
250	250	121.1/23.41 2.694/10.97	406.5/57.50 10.34/30.09	340.5/48.86 7.606/22.03
300	300	192.7/40.65 3.105/13.59	674.2/123.7 13.10/38.76	626.2/89.60 9.526/27.99
350	350	*/63.40 3.482/15.77	*/148.9 12.43/36.74	*/143.1 10.89/32.15
400	400	*/92.14 3.740/17.29	*/259.9 15.97/47.78	*/209.6 12.12/36.19
450	450	*/129.9 4.019/18.97	*/390.2 17.85/59.04	*/310.9 13.78/41.61
500	500	*/189.0 4.664/22.52	*/665.0 22.71/69.45	*/461.8 16.68/51.13

Table 4. Experiments with NETGEN problems. The total supply used was 10^6, and the capacity range was used [1, 1000].

Nodes	Arcs	AUCTION-MF	PFP-AO	PFP-DM
1000	10000	24.35 21.05/101.0	72.90 42.34/197.6	67.05 21.59/132.8
2000	20000	84.17 32.46/163.3	170.6 38.89/210.9	219.5 29.84/201.1
3000	30000	121.3 27.28/154.6	263.7 35.31/204.5	320.0 28.04/203.1
4000	40000	159.2 25.46/146.4	371.3 33.43/205.4	443.4 26.83/206.1
5000	50000	202.7 24.31/146.3	482.2 32.05/207.8	569.7 25.87/209.2

Figures 3–5 show the factor of superiority of AUCTION-MF over the preflow-push codes on RMFGEN and GRID-SQ problems in terms of computation time. The figures indicate that this factor tends to increase with problem dimension, particularly for the square grid problems.

6. Conclusions

There are two main methodological conclusions of this paper:

(i) Using shortest path augmentations within the Ford–Fulkerson framework is not essential for good worst-case or practical performance.

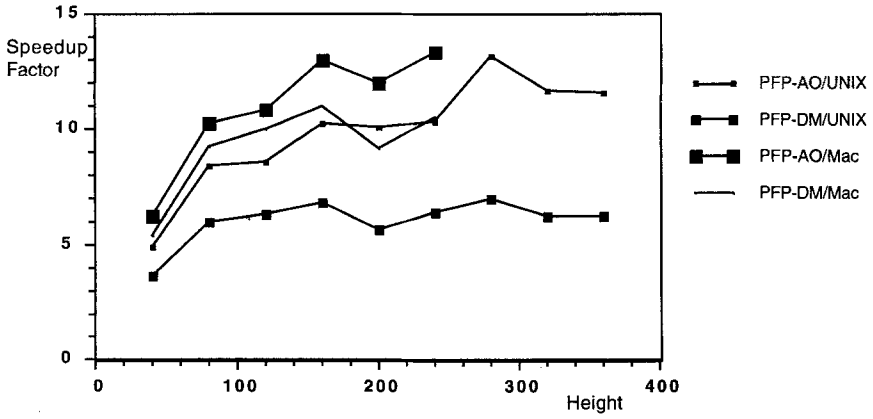

Fig. 3. Speedup factor of AUCTION-MF over the PFP-AO and PFP-DM codes for RMFGEN problems with constant side =15. Compare with Table 1.

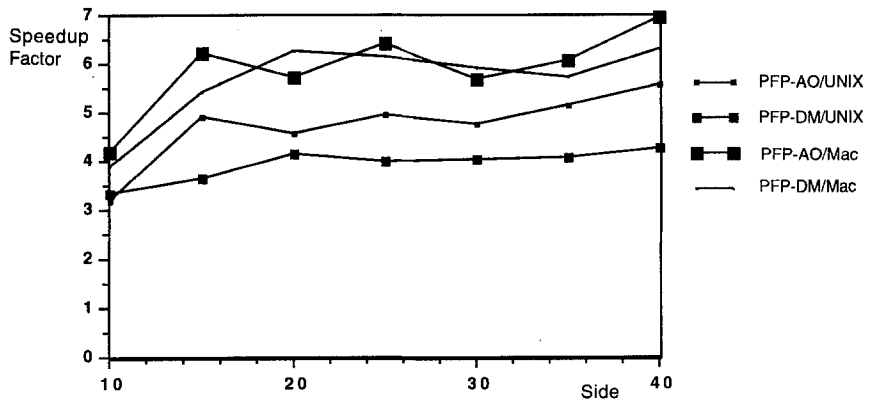

Fig. 4. Speedup factor of AUCTION-MF over the preflow-push codes for RMFGEN problems with constant height =40. Compare with Table 2.

Instead, it is important to transfer efficiently useful information from one augmenting path construction to the next. The prices of the auction algorithm provide an effective mechanism for such a transfer.

(ii) The augmenting path approach, when properly implemented through the use of a path construction algorithm based on price adjustment and auction ideas, can substantially outperform the preflow-push approach. This contradicts the current mainstream thinking in the field, which following extensive recent numerical experimentation, considers preflow-push methods as superior to augmenting path methods.

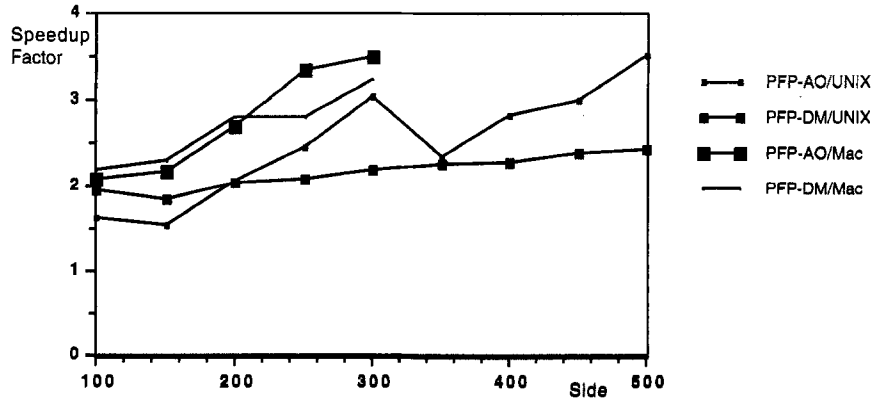

Fig. 5. Speedup factor of AUCTION-MF over the preflow-push codes for GRID-SQ problems. Compare with Table 3.

The new max-flow algorithm given in this paper is supported by strong computational evidence. It is substantially faster than preflow-push methods on standard randomly generated problems, and tends to perform far fewer flow changes and price changes. What is happening here is that, in our method, arc flows change only after the node prices have risen to the proper level for an augmentation, whereas in preflow-push methods flows change simultaneously, and often unnecessarily, with the prices. Deferring flow changes until an augmentation can be performed has an additional side effect: it does not disturb the reduced graph unnecessarily, thereby confusing the search for an augmenting path. This, together with some additional special features of our method, such as performing extensions along arcs of Cand(i) which are not strictly downhill, explains the experimentally observed large reduction in the number of price changes over preflow-push methods. It is generally thought that the larger flow increments resulting from the use of single-arc versus multiple-arc paths in preflow-push methods is a significant advantage. However, it appears that the use of greedy augmentations nullifies to a large extent this perceived advantage. Furthermore, the inferiority of the worst-case running time of our method relative to the one of the best preflow-push methods is of little practical significance, because the practical computational bottleneck is the work for price increases, which is comparable [$O(NA)$] for both methods.

The ideas of the present paper admit extension to minimum cost flow problems along the lines of the auction/sequential shortest path algorithm developed in Ref. 22. One may simply substitute the auction algorithm for constructing shortest augmenting paths of Ref. 22 with the simpler path construction algorithms used here. Results using this approach will be reported elsewhere.

7. Appendix: Relation of the Path Construction Algorithm and the Auction Algorithm

Assuming that the reader is familiar with the auction algorithm for the assignment problem, as given, for example, in Ref. 5 or Ref. 29, we will draw the connection of the auction algorithm with the first path construction algorithm of Section 2. To this end, we note that the path construction problem can be converted into a pure, unweighted matching problem as shown in Fig. 6. In particular, each arc (i,j) of the reduced graph $\mathcal{G}$, with $i \neq n_1$, N, is replaced by an object labeled (i,j). Each node $i \neq N$ is replaced by $R(i)$ persons, where $R(i)$ is the number of arcs of $\mathcal{G}$ that are incoming to node i; for example, in Fig. 6, node 2 is replaced by the two persons 2 and 2′. Finally, there is one person corresponding to node n_1 and one object

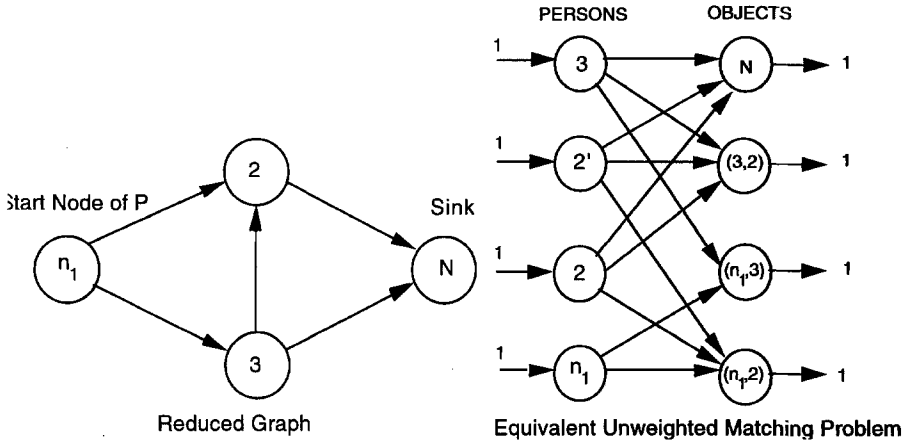

Fig. 6. Converting the augmenting path construction problem into an equivalent problem of (unweighted) matching of "persons" to "objects."

corresponding to node N. For every arc (i,j) of $\mathcal{G}$, with $i \neq N$, there are $R(i)+R(j)$ incoming arcs from the persons corresponding to i and j. For every arc (i, N) of $\mathcal{G}$, there are $R(i)$ incoming arcs from the persons corresponding to i.

Each path that starts at n_1 and ends at N can be associated with a feasible matching. For example, in Fig. 6, the path $(n_1, 3, 2, N)$ corresponds to the feasible matching

$$(n_1, (n_1, 3)), \quad (3, (3, 2)), \quad (2, (2, N)), \quad (2', (n_1, 2)),$$

or the same pairs with the roles of 2 and 2′ interchanged. Conversely, given a feasible matching, one can construct an alternating path (a sequence of alternatively assigned and unassigned pairs) starting at n_1 and ending at N, which defines a path from n_1 to N. For example, in Fig. 6, the feasible matching comprising the pairs

$$(n_1, (n_1, 3)), \quad (3, (3, N)), \quad (2, (n_1, 2)), \quad (2', (3, 2))$$

corresponds to the path $(n_1, 3, N)$.

A set of valid prices for the nodes of $\mathcal{G}$ defines the prices of the objects of the matching problem by

$$r_{ij} = p(i), \qquad \forall (i,j) \in \mathcal{A},$$

$$r_N = p(N).$$

These prices together with the incomplete matching that pairs a person corresponding to node j with some arc (i,j) incoming to j, satisfy the ϵ-complementary slackness condition of the auction algorithm with $\epsilon = 1$. In such a matching, the only unassigned person is the one corresponding to node n_1, and the only unassigned object is the one corresponding to node N. If we apply the auction algorithm with $\epsilon = 1$, starting from this matching-price pair, it can be verified that the sequence of generated prices and matchings correspond to the sequence of prices and paths generated by the first path construction algorithm of Section 2.

References

1. BERTSEKAS, D. P., *An Auction Algorithm for the Max-Flow Problem*, Report LIDS-P-2193, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1993.
2. FORD, L. R., JR., and FULKERSON, D. R., *Maximal Flow through a Network*, Canadian Journal of Mathematics, Vol. 8, pp. 339–404, 1956.
3. FORD, L. R., JR., and FULKERSON, D. R., *Flows in Networks*, Princeton University Press, Princeton, New Jersey, 1962.
4. PAPADIMITRIOU, C. H., and STEIGLITZ, K., *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
5. BERTSEKAS, D. P., *Linear Network Optimization: Algorithms and Codes*, MIT Press, Cambridge, Massachusetts, 1991.
6. EDMONDS, J., and KARP, R. M., *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*, Journal of the ACM, Vol. 19, pp. 248–264, 1972.
7. DINIC, E. A., *Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation*, Soviet Mathematics Doklady, Vol. 11, pp. 1277–1280, 1970.
8. BERTSEKAS, D. P., *A Distributed Algorithm for the Assignment Problem*, Working Paper, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1979.
9. KARZANOV, A. V., *Determining the Maximal Flow in a Network with the Method of Preflows*, Soviet Mathematics Doklady, Vol. 15, pp. 434–437, 1974.
10. SHILOACH, Y., and VISHKIN, U., *An $O(n^2 \log n)$ Parallel Max-Flow Algorithm*, Journal of Algorithms, Vol. 3, pp. 128–146, 1982.
11. GOLDBERG, A. V., *A New Max-Flow Algorithm*, Technical Memorandum MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1985.
12. GOLDBERG, A. V., and TARJAN, R. E., *A New Approach to the Maximum Flow Problem*, Proceedings of the 18th ACM STOC, pp. 136–146, 1986.
13. AHUJA, R. K., and ORLIN, J. B., *A Fast and Simple Algorithm for the Maximum Flow Problem*, Operations Research, Vol. 37, pp. 748–759, 1989.

14. AHUJA, R. K., MAGNANTI, T. L., and ORLIN, J. B., *Network Flows*, Handbooks in Operations Research and Management Science, Vol. 1; Optimization, Edited by G. L. Nemhauser, A. H. G. Rinnooy-Han, and M. J. Todd, North Holland, Amsterdam, Holland, pp. 211–369, 1989.
15. CHERIYAN, J., and MAHESHWARI, S. N., *Analysis of Preflow-Push Algorithms for Maximum Network Flow*, SIAM Journal on Computing, Vol. 18, pp. 1057–1086, 1989.
16. DERIGS, U., and MEIER, W., *Implementing Goldberg's Max-Flow Algorithm: A Computational Investigation*, Zeitschrift für Operations Research, Vol. 33, pp. 383–403, 1989.
17. MAZZONI, G., PALLOTINO, S., and SCUTELLA', M. G., *The Maximum Flow Problem: A Max-Preflow Approach*, European Journal of Operational Research, Vol. 53, pp. 257–278, 1991.
18. BERTSEKAS, D. P., *Mathematical Equivalence of the Auction Algorithm for Assignment and the ϵ-Relaxation (Preflow-Push) Method*, Large-Scale Optimization: State of the Art, Edited by W. W. Hager, D. W. Hearn, and P. M. Pardalos, Kluwer Academic Publishers, Amsterdam, Holland, pp. 26–44, 1994.
19. ANDERSON, R. J., and SETUBAL, J. C., *Goldberg's Algorithm for Maximum Flow in Perspective: A Computational Study*, Algorithms for Network Flows and Matching, Edited by D. S. Johnson and C. C. McGeoch, American Mathematical Society, Providence, Rhode Island, pp. 1–18, 1993.
20. NGUYEN, Q. C., and VENKATESWARAN, V., *Implementations of the Goldberg–Tarjan Maximum Flow Algorithm*, Algorithms for Network Flows and Matching, Edited by D. S. Johnson and C. C. McGeoch, American Mathematical Society, Providence, Rhode Island, pp. 19–41, 1993.
21. BERTSEKAS, D. P., *The Auction Algorithm for Shortest Paths*, SIAM Journal on Optimization, Vol. 1, pp. 425–447, 1991.
22. BERTSEKAS, D. P., *An Auction/Sequential Shortest Path Algorithm for the Min Cost Flow Problem*, Report LIDS-P-2146, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1992.
23. CASTANON, D. A., *Reverse Auction Algorithms for Assignment Problems*, Algorithms for Network Flows and Matching, Edited by D. S. Johnson and C. C. McGeoch, American Mathematical Society, Providence, Rhode Island, pp. 407–429, 1993.
24. BERTSEKAS, D. P., PALLOTTINO, S., and SCUTELLA', M. G., *Polynomial Auction Algorithms for Shortest Paths*, Report LIDS-P-2107, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1992.
25. GOLDFARB, D., and GRIGORIADIS, M. D., *A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow*, Fortran Codes for Network Optimization, Edited by B. Simeone et al., Annals of Operations Research, Vol. 13, pp. 83–123, 1988.
26. KLINGMAN, D., NAPIER, A., and STUTZ, J., *NETGEN: A Program for Generating Large-Scale (Un) Capacitated Assignment, Transportation, and Minimum Cost Flow Network Problems*, Management Science, Vol. 20, pp. 814–822, 1974.

27. BERTSEKAS, D. P., and TSENG, P., *RELAX: A Computer Code for Minimum Cost Network Flow Problems*, Fortran Codes for Network Optimization, Edited by B. Simeone et al., Annals of Operations Research, Vol. 13, pp. 127–190, 1988.
28. CAPITANI, G., CATONI, O., and GALLO, G., Private Demonstration and Communication, 1994.
29. BERTSEKAS, D. P., *Auction Algorithms for Network Flow Problems: A Tutorial Introduction*, Computational Optimization and Applications, Vol. 1, pp. 7–66, 1992.