JOURNAL OF ALGORITHMS 17, 424–446 (1994)

A Faster Algorithm for Finding the Minimum Cut in a Directed Graph

JIANXIU HAO

GTE Laboratories, Waltham, Massachusetts, 02154

AND

JAMES B. ORLIN

Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

Received January 10, 1992; revised October 16, 1992

We consider the problem of finding the minimum capacity cut in a directed network G with n nodes. This problem has applications to network reliability and survivability and is useful in subroutines for other network optimization problems. One can use a maximum flow problem to find a minimum cut separating a designated source node s from a designated sink node t, and by varying the sink node one can find a minimum cut in G as a sequence of at most $2n - 2$ maximum flow problems. We then show how to reduce the running time of these $2n - 2$ maximum flow algorithms to the running time for solving a single maximum flow problem. The resulting running time is $O(nm \log(n^2/m))$ for finding the minimum cut in either a directed or an undirected network. © 1994 Academic Press, Inc.

1. INTRODUCTION

A classic result in the area of network flows is that the maximum flow from a designated source node s to a designated sink node t is equal to the minimum capacity of a cut separating s from t. This simple but elegant result, proved by Ford and Fulkerson [6], has a number of applications in practice, and unifies a number of disparate results in the area of combinatorial optimization.

In this paper, we focus on finding a minimum capacity cut in a directed network in which no source node or sink node is specified. (To apply this algorithm to undirected networks, one can replace each undirected arc (i, j) with two directed arcs (i, j), and (j, i), each with capacity u_{ij}). The classical maximum flow problem assumes the existence of a designated source node s and sink node t. We refer to any cut $(S, N - S)$ with $s \in S$ and $t \in N - S$ as an s–t cut, and we refer to the problem of determining a minimum capacity s–t cut as the *s–t cut problem*. Although the s–t cut problem has a wide range of applications (see, for example, [1, 15] for a number of these applications), for other applications one wants to determine the minimum cut in a network in which there is no designated source or sink node. In this case, one wants to partition the node set N of a network into two nonempty parts denoted as S^* and $N - S^*$ such that the capacity of the cut $(S^*, N - S^*)$ is as small as possible. We refer to this problem as the "minimum unrestricted cut problem" to emphasize that there are no designated source and sink nodes.

In the case where the network is directed, the previous best algorithm for solving the minimum unrestricted cut problem (in terms of performance guarantees) was to solve the problem as a sequence of $2n - 2$ minimum s–t cut problems (find the minimum s–j cut and the minimum j–s cut for each $j \neq s$ and select the best of these $2n - 2$ cuts), and thus the running time was $O(nM(n, m))$, where $M(n, m)$ is the best running time for solving a maximum flow problem with n nodes and m arcs. Currently the best strongly polynomial running time algorithms for solving the maximum flow problem are a randomized algorithm developed by Cheriyan *et al.* [4] and a deterministic $O(\min(nm + n^{2+\varepsilon}, nm \log n))$ time algorithm for any fixed $\varepsilon > 0$ developed by King *et al.* [9]. Thus, the minimum directed cut algorithm presented in this paper improves upon the previous best algorithm by nearly a factor of n.

In the case that the network is undirected, the best algorithm for solving the minimum unrestricted cut problem has a running time of $O(nm + n^2 \log n)$ and was developed by Nagamochi and Ibaraki [12]. See also Nagamochi and Ibaraki [13] for an interesting related result. The Nagamochi–Ibaraki algorithm dominates our algorithm by a factor of $\log n$ except for very dense networks. Of course, the performance in the worst case does not necessarily indicate the performance in practice. The algorithm which is the better in practice would need to be determined through appropriate computational testing. An interesting special case is the one in which all arcs have a capacity of 1. This case has been studied by several authors including Matula [10] and Mansour and Schieber [11].

In this paper, we show how to solve the minimum cut problem as a sequence of $2n - 2$ minimum s–t cut problems in such a way that the total running time is comparable to the time to solve one minimum $s - t$ cut problem. In particular, one can solve all of the $O(n)$ minimum s–t cut problems in a total of $O(nm \log(n^2/m))$ steps. The algorithm relies on a simple idea, also exploited in the very efficient parametric flow algorithm developed by Gallo *et al.* [7]: the initial distance labels for any minimum

s–t cut problem are always the distance labels obtained by the algorithm at the termination of solving the preceding minimum s–t cut problem. (We describe distance labels in the next section.) If one handles distance labels carefully, and if one solves the $O(n)$ minimum cut problems in an appropriate order, then the running time to solve the $O(n)$ minimum cut problems using a variant of the Goldberg–Tarjan preflow push algorithm is comparable to the time to find a single minimum s–t cut. We remark that the best weakly polynomial algorithms for the minimum s–t cut problem, as well as the algorithm by King, Rao, and Tarjan, rely on the idea of scaling, and these ideas do not seem to extend to our approach for solving the minimum unrestricted cut problem.

The algorithm can also be sped up in the case that the network is bipartite, using the algorithm of Ahuja *et al.* [2]. If the number of nodes on the smaller half of the bipartite network is n', and if n' is much smaller than n, then the running time of the algorithm can be improved to $O(n'm \log((2 + n'^2/m))$.

Before proceeding to a technical description of the results, we first give two motivating applications. The first application is to find the most unreliable cut in a directed network. Consider a directed network $G = (N, A)$ in which the probability of any arc (i, j) working reliably is p_{ij}. Ideally, p_{ij} is close to one, and for every pair i, j of nodes, there is some reliable directed path from i to j; however, in practice many arcs may be unreliable, and they are susceptible to failure. If there is no working path from i to j, then there is some cut $(S, N - S)$ separating i from j such that $i \in S$, $j \in N - S$, and there is no arc directed from any node of S to any node in $N - S$. The probability that each arc in the cut $(S, N - S)$ is not working is $F(S, N - S) = \prod_{i \in N,\ j \in N-S}(1 - p_{ij})$. Suppose that one wants to identify the directed cut in G which is the most likely to fail. Such a cut is useful in diagnosing the susceptibility of the network to failure, and may also be of use in determining where the network should be fortified.

To maximize the probability of failure is equivalent to minimizing $-\log F(S, N - S)$, which is $\sum_{i \in N,\ j \in N-S} - \log(1 - p_{ij})$. So if we let $u_{ij} = -\log(1 - p_{ij})$, a minimum directed cut in the network corresponds to a directed cut with maximum probability of failure.

We now consider another application, described more fully in the paper by Padberg and Rinaldi [14]. Consider the linear programming relaxation of the traveling salesman problem in which one has assignment constraints and subtour elimination constraints. Suppose that x is a solution that satisfies the assignment constraints, but one wants to know whether it also satisfies the subtour elimination constraints. Equivalently, one wants to determine whether there is a partition of the nodes into two parts, say S^* and $N - S^*$ such that $\sum_{i \in S^*,\ j \in N-S^*} x_{ij} < 1$. If we interpret the flows x_{ij} as capacities for a minimum cut problem, one wants to know whether the capacity of a minimum cut in the network is less than 1. Padberg and Rinaldi developed an approach that is very efficient in practice, but has a worst case running time comparable to that of solving n maximum flow problems. Our approach solves the problem in the time to solve a single maximum flow problem. It is plausible that our approach is as efficient or more efficient than the algorithm developed by Padberg and Rinaldi; however, the relative efficiency of these two approaches can only be assessed by detailed computational testing.

2. Notation, Definitions, and an Overview of the Algorithm

For the most part, the notation in this paper is consistent with the notation in the text by Ahuja *et al.* [1]. We refer the reader to that text for further details on notation, and for further details on the algorithm developed by Goldberg and Tarjan [8].

Let $G = (N, A)$ be a directed network with node set N and arc set A. Let n and m denote the number of nodes and arcs of the network respectively. Each arc of the network has an associated capacity u_{ij}, which is a nonnegative real number. (In practice, one would assume that u_{ij} is a rational number, but since this algorithm is strongly polynomial, we do not need this additional assumption.) A *cut* refers to a partition of the node set into two nonempty parts S and $N - S$. The *capacity* of the cut $(S, N - S)$ is $\sum_{i \in S,\, j \in N-S} u_{ij}$, and we denote the capacity of the cut as $u(S, N - S)$. The *minimum unrestricted cut problem* is to find a partition of N into two nonempty parts, S^* and $N - S^*$, so as to minimize $u(S^*, N - S^*)$.

Suppose that S is a subset of nodes and j is a single node. The *minimum S–j cut problem* is the following:

$$\text{minimize } \{u(S^*, N - S^*) : S \subseteq S^*, \text{ and } j \in N - S^*\}.$$

In other words, one wants a minimum cut subject to the additional condition that the source side of the cut contains the subset S and the sink side of the cut contains node j.

In the algorithm below we will determine the minimum cut $(S^*, N - S^*)$ in the network subject to the condition that a designated node s is in S^*. If we apply the algorithm to the graph obtained from G by reorienting each of the arcs, then we obtain the minimum cut $(N - T^*, T^*)$ with the property that $s \in T^*$; i.e., s is on the sink side of the cut. The minimum cut in the network is the better of the two cuts $(S^*, N - S^*)$ and $(N - T^*, T^*)$.

Henceforth, we will focus on the problem of identifying the minimum cut $(S^*, N - S^*)$ with the property that s is on the source side of the cut.

The previous argument shows that this algorithm can be used to solve the minimum unrestricted cut problem as well. Our algorithm, as presented at its most abstract and highest level, is essentially the same as the Padberg and Rinaldi algorithm. However, the algorithm differs at a more detailed level, and these differences account for the improved running time.

```
Algorithm FindMinCut(s)
begin
  $S := \{s\}$;
  BestValue := $\infty$;
  while $S \neq N$ do
    begin
      select some node $t' \in N - S$;
      determine a minimum $S$–$t'$ cut $(S^*, N - S^*)$;
      $z := u(S^*, N - S^*)$;
      if $z <$ BestValue, then Cut := $(S^*, N - S^*)$ and BestValue := $z$;
      add $t'$ to $S$;
    end
```

The description of the algorithm *FindMinCut*( ) shows how to solve the minimum cut problem as a sequence of $O(n)$ minimum S–t' cut problems. The algorithm uses the notation t' to emphasize that the sink node t' changes from one minimum cut problem to the next. We next claim that the algorithm does indeed determine a minimum cut with s on the source side.

LEMMA 1. *The algorithm FindMinCut(s) determines a minimum capacity cut with the property that s is on the source of the cut.*

Proof. Let $(S', N - S')$ denote the minimum capacity cut in the network with the property that s is on the source side, and let $(S^*, N - S^*)$ denote the cut obtained by the algorithm. We first note that $(S^*, N - S^*)$ is a feasible cut with $s \in S^*$, and so $u(S^*, N - S^*) \geq u(S', N - S')$. We next show that $u(S^*, N - S^*) \leq u(S', N - S')$ as well. Let us suppose without loss of generality that $s = 1$, and the remaining nodes are labeled $2, 3, 4, \ldots, n$ in the order of their selection as sinks by the minimum cut algorithm. Let j be the minimum index of a node in $N - S'$. At the time that the algorithm selects j as the sink node, $S = \{1, \ldots, j - 1\}$. By our choice of the node j, S is a subset of S'. Thus $(S', N - S')$ is a feasible S–j cut, and thus the objective value of the minimum S–j cut is at most $u(S', N - S')$, proving the result. ■

The algorithm *FindMinCut*(s) solves the minimum unrestricted cut problem as a sequence of $O(n)$ minimum $S - t'$ cut problems. However, we will soon show that if we are careful in the order in which we select the sink nodes, and if we modify the generic preflow push algorithm appropriately, then in the worst case we can solve these $n - 1$ minimum $S - t'$ cut problems in the time that it takes in the worst case to solve just one minimum $s - t$ cut problem. Using dynamic trees, the running time is $O(nm \log n^2/m)$, as is the running time of Goldberg and Tarjan's [8] algorithm for the maximum flow and minimum s–t cut problems. Using highest level pushing and without using dynamic trees, the running time is $O(n^2 m^{1/2})$. This running time is comparable to the running time of the Goldberg–Tarjan preflow push algorithm with highest level pushing, as proved by Cheriyan and Maheshwari [3].

In order to achieve the improvement in the running time, one needs to modify the steps of the preflow push algorithm appropriately. In the next section we describe the modifications of the preflow push algorithm.

3. The Preflow-Push Algorithm

In this section, we describe the preflow push algorithm for finding a minimum S–t cut, where S is a designated subset of nodes and t is a designated sink node. Our algorithm differs from the usual preflow push algorithm in a couple of ways. First of all, the normal preflow push algorithm finds a minimum s–t cut for some source node s rather than an S–t cut; this distinction, however, is rather minor since it is easy to transform the S–t cut problem into an s–t cut problem. More significantly, our algorithm partitions the node set N into two parts W and $D = N - W$, where D is a set of "dormant nodes" and W is a set of nodes that are "awake." The algorithm maintains the invariant that there is no arc of the "residual network" directed from any node in D to any node in W.

In Section 4, we will show how to modify the preflow-push algorithm of this section to solve the minimum cut problem. We have modified the preflow push algorithm in this section specifically so that it can be extended to the minimum cut algorithm of the next section. In particular, the rationale for partitioning the nodes into dormant and awake nodes will be made apparent in the next section.

Let x be any m-vector indexed on the arcs of A such that $0 \le x_{ij} \le u_{ij}$ for each $(i, j) \in A$. Recall that S is the set of source nodes. For each node $i \in N$, the *excess* of node i is denoted as $e(i)$ and is defined as follows:

$$e(i) = \sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} \qquad \text{for all } i \in N - S.$$

We say that x is a *preflow* if $0 \le x_{ij} \le u_{ij}$ for each $(i, j) \in A$ and if $e(i) \ge 0$ for each $i \in N - S$; i.e., a preflow satisfies the bound constraints, and the net flow into node i exceeds the outflow except for source nodes.

We assume that for every arc $(i, j) \in A$, there is also an arc (j, i) possibly with 0 capacity. This assumption is without loss of generality, and is made solely for notational convenience. Given a preflow x, the *residual capacity* r_{ij} of any arc $(i, j) \in A$ is the maximum additional flow that can be sent from node i to node j using the arcs (i, j) and (j, i). The residual capacity r_{ij} has two components: (i) $u_{ij} - x_{ij}$, the unused capacity of arc (i, j), and (ii) the current flow x_{ji} on arc (j, i), which we can cancel to increase the flow from node i to node j. Consequently, $r_{ij} = u_{ij} - x_{ij} + x_{ji}$. We refer to the network $G(x) = (N, A(x))$ consisting of the arcs with positive residual capacities as the *residual network* with respect to the flow x.

We will partition the nodes of the network into two parts, W and D, denoting respectively the set of nodes that are *awake* and the set of nodes that are *dormant*. Each node i has a distance label $d(i)$ which is an integer in the range $\{0, 1, \dots, n\}$. (We remark that for maximum flow problems the range is usually $\{0, \dots, 2n - 1\}$; however, the range may be made smaller for minimum cut problems.) In this section, we will generally assume that $d(t) = 0$; however, in the next section, we will relax this assumption and allow $d(t)$ to take on other integer values between 1 and n.

We say that the set of distance labels are *W-valid* if they satisfy the following property:

W-Validity Property. For each pair of nodes i and j in W, if $r_{ij} > 0$ then $d(i) \leq d(j) + 1$.

In addition, the nodes of D will satisfy the following property:

Dormancy Property. We say that nodes in D satisfy the *dormancy property* if $t \notin D$ and for each node $i \in D$ there is no arc $(i, j) \in G(x)$ with $j \in W$.

As a result, if node i is in D and if D satisfies the dormancy property, then there can be no path in the residual network $G(x)$ from i to t or to any other node in W. The following Lemma is a variation of a lemma given in Goldberg and Tarjan [8], and is easily proved via induction.

LEMMA 2. *Suppose that the set $d(\ )$ of distance labels is W-valid and that the set $D = N - W$ satisfies the dormancy property. Then for each node i, $d(i)$ is a lower bound on the number of arcs in any path from i to t in $G(x)$.*

A node i said to be *active* if $i \in W - \{t\}$ and if $e(i) > 0$. An arc (i, j) is said to be *admissible* if $i \in W$, $j \in W$, $d(i) = d(j) + 1$, and $r_{ij} > 0$. In general, the algorithm will send flow from active nodes and will send flow along admissible arcs. We are now ready to give the preflow push algorithm.

```
Algorithm preflow-push;
begin
   Initialize;
   while the network contains an active node do
     begin
       select an active node $i$;
       if the network contains an admissible arc $(i, j)$ then
          push $\delta := \min\{e(i), r_{ij}\}$ units of flow from node $i$ to node $j$;
       else relabel($i$);
     end;
end;

Procedure Initialize
begin
   for each arc $(i, j)$ with $i \in S$ and $j \in N - S$, send $r_{ij}$ units of flow in
      $(i, j)$;
   $D := S$;
   $W := N - S$;
   $d(t) := 0$;
   for each node $j \in N - \{t\}$ do $d(j) := 1$; {alternatively, the distance
    label of  node $j$ can be computed to be the minimum number of
    arcs
     no a path from $j$ to $t$}
end

Procedure relabel($i$);
begin
   if $i$ is the only node in $W$ with distance label $d(i)$ then
        begin
          $R := \{j \in W : d(j) \geq d(i)\}$;
          $D := D \cup R$;
          $W := W - R$;
        end
   else if there is no arc $(i, j)$ in $G(x)$ with $j \in W$, then $D := D \cup \{i\}$ and
      $W := W - \{i\}$;
   else $d(i) := \min\{d(j) + 1 : (i, j) \in A(i), j \in W$ and $r_{ij} > 0\}$;
end;
```

We now briefly outline the differences between the usual implementation of the G–T preflow push algorithm and the algorithm presented above. Later, we will modify the algorithm further so that it can solve the minimum unrestricted cut problem, and we will prove the correctness of our minimum unrestricted cut algorithm in Section 4.

As stated in the introduction to this section, the algorithm partitions the node set into two disjoint subsets, D and $W = N–D$. The set D refers to the set of dormant nodes, and there is no arc in the residual network from any node in D to any node in W. At initialization, the algorithm saturates all of the arcs emanating from nodes in S; at this point, there is no further arc in the residual network from a node in S to a node in $N - S$, and the algorithm initializes D as the set S and W as the set $N - S$.

The algorithm performs pushes in a similar manner to the $G–T$ algorithm; however, the $G–T$ algorithm permits any node with positive excess to be selected for pushing, whereas the algorithm presented here enforces the additional requirement that the selected node be in W. Nodes in W with positive excess are called "active." Also, no flow is permitted to be pushed from a node i in W to a node j in D, even if $d(i) = d(j) + 1$.

The algorithm performs relabels in a similar manner to the $G–T$ algorithm, except for two cases. In the first case, the algorithm identifies an active node i that needs to be relabeled but with the property that no other node has distance label $d(i)$. Then the algorithm lets $R = \{j \in W : d(j) \geq d(i)\}$, and it transfers the nodes of R from W to D. As stated in the next lemma, there is no arc directed from a node in R to a node in $W - R$, and so the modified set D will still satisfy the dormancy property, assuming that D satisfied the dormancy property prior to the transfer of the set R.

LEMMA 3. *Suppose that the set $d(\ )$ of distance labels is W-valid and that the set $D = N–W$ satisfies the dormancy property. Suppose in addition that i has no admissible arcs, and there is no other node in W with distance label $d(i)$. Let $R = \{j \in W : d(j) \geq d(i)\}$. Then there is no arc in the residual network directed from a node in R to a node in W–R. In addition, the set $D' = D \cup R$ satisfies the dormancy property.*

Proof. By assumption and by the definition of the set R, there is no arc directed from node i to a node in $W - R$. Let j be any other node in R. By assumption, $d(j) > d(i)$. Since $j \in W$, and since the distance labels are W-valid, if follows for any node $k \in W$ and for any arc (j, k) in the residual network that $d(k) \geq d(j) - 1 \geq d(i)$. Hence $k \in R$. We conclude that there is no arc directed from a node in R to a node in $W - R$. If we let $D' = D \cup R$, it follows that there is no arc directed from a node in D' to a node in $W - R$, and thus the lemma is true. ■

We note that the procedure for identifying dormant nodes was presented as a heuristic for speeding up the computations in Derigs and Meier [5]. It is also described in Ahuja *et al.* [1], where it is called the "gap heuristic." The primary difference here is that the nodes are identified as dormant, whereas in usual implementations of the G–T algorithm, these nodes are assigned a distance label of n.

There is one other situation in which a dormant node is identified. Suppose that node i needs to be relabeled, and there is no arc of the residual network directed from i to a node in W. In this case, we can add i to D and delete it from W. It is easy to see that the resulting set $D \cup \{i\}$ satisfies the dormancy property so long as D satisfies the dormancy property.

We have now completed the review of the G–T preflow push algorithm. In the next section, we show how to modify the algorithm further so as to efficiently solve the minimum cut problem. Subsequently, we will prove the correctness of the algorithm and its time bounds.

4. The Minimum Cut Algorithm

To start off this section we expand our previous outline of the procedure *FindMinCut*(s).

```
Algorithm FindMinCut($s$)
begin
  ModifiedInitialize
  while $S \neq N$ do
    begin
      while the network contains an active node do
        begin
          select an active node $i$;
          if the network contains an admissible arc $(i, j)$ then
            push $\delta := \min\{e(i), r_{ij}\}$ units of flow from node $i$ to
              node $j$;
          else ModifiedRelabel($i$);
        end;
      —Recall that $D$ denotes the set of dormant nodes, and $W =$
        $N - D$—
      if BestValue $> u(D, W)$, then Cut $:= (D, W)$ and BestValue $=:$
        $u(D, W)$;
      SelectNewSink;
    end
end
```

The algorithm has as its inner loop the preflow push algorithm of the previous section with some small but technically needed differences. We now summarize these differences, and subsequently we will prove the correctness of the algorithm.

In the procedure *ModifiedRelabel*, the algorithm will relabel a node i or else it will transfer i and possibly other nodes of W to the set D of dormant nodes. The difference between *ModifiedRelabel* and *Relabel* is that in *ModifiedRelabel* we will partition the set D into subsets called DormantSet(0), DormantSet(1), ..., DormantSet($D_{\max}$) according to the step at which the nodes were transferred from W to D. For example, suppose at some iteration that $D_{\max} = K$ and that $D = \bigcup_{i=0}^{K}$DormantSet(i). Suppose at that iteration that one wants to transfer the set R of nodes from W to D. This is accomplished by incrementing $D_{\max}$ from K to $K + 1$, and then letting DormantSet($K + 1$) $= R$. DormantSet(i) is, in general, determined by transferring a subset of nodes from W to D in a relabel, except for DormantSet(0) which is the set S of source nodes.

As in the description of the preflow push algorithm in Section 3, there is no arc directed from a node in D to a node in W; so the nodes in D need not be considered in the current minimum $S - t'$ cut procedure; however, these nodes may need to be considered at a later stage in the algorithm *FindMinCut* since the algorithm solves a sequence of $S - t'$ minimum cut problems, with each node being the sink node once except for node s. After determining a minimum $S - t'$ cut, the algorithm then transfers t' to S, and selects a new sink node. If possible, it selects the node in W with minimum distance label; however, it is possible that t' was the only node in W at the end of the procedure for finding a minimum $S - t'$ cut, and that subsequent of the transfer of t' to S, there is no node in W. If W is empty, the algorithm then transfers the set DormantSet($D_{\max}$) to W, which makes W nonempty again. Once W is nonempty, the algorithm selects the node in W with minimum distance label, and starts a new minimum cut procedure.

Each of the dormant nodes, except for the nodes in S, will ultimately be "awakened" and transferred back to W. In fact, each of these nodes will be the sink node for some subsequent minimum cut problem.

We now provide the remaining procedures and prove the correctness of the algorithm. We will assume that node 1 is the source node and that node 2 is the initial sink node.

```
Procedure ModifiedInitialize
begin
    for each arc $(1, j)$ send $r_{1j}$ units of flow in $(1, j)$;
    DormantSet(0) := {1};
    $D_{\max}$ := 0;
    $W := N - \{1\}$;
    $t' := 2$;
    $d(t') := 0$;
    for each node $j \in N - \{t'\}$ do $d(j) := 1$;
end

Procedure ModifiedRelabel($i$);
begin
    if $i$ is the only node in $W$ with distance label $d(i)$ then
        begin
            $D_{\max} := D_{\max} + 1$;
            $R := \{j \in W : d(j) \geq d(i)\}$;
            DormantSet($D_{\max}$) $:= R$;
            $W := W - R$;
        end
    else if there is no arc $(i, j)$ in $G(x)$ with $j \in W$, then
        begin
            $D_{\max} := D_{\max} + 1$;
            DormantSet($D_{\max}$) $:= \{i\}$;
            $W := W - \{i\}$;
        end
    else $d(i) := \min\{d(j) + 1 : (i, j) \in A(i),\ j \in W$ and $r_{ij} > 0\}$;
end;

Procedure SelectNewSink
begin
    —let node $t'$ denote the old sink node—
    delete $t'$ from $W$;
    add $t'$ to both set $S$ and to DormantSet(0);
    if $S = N$, then quit; else continue;
    for each arc $(t', k)$ with $k \in N - S$ do send $r_{t'k}$ units of flow in arc
$(t', k)$;
    if $W = \varnothing$ then
        begin
            $W :=$ DormantSet($D_{\max}$);
            $D_{\max} := D_{\max} - 1$;
        end
    select $j \in W$ such that $d(j)$ is minimum;
    $t' := j$;
end
```

The correctness of the algorithm and its time bounds rely on several properties satisfied by the algorithm. We list these properties below, and prove them subsequently. For a given set V of nodes, we let $d(V) =$

$\{k : d(j) = k$ for some $j \in V\}$. In other words, $d(V)$ is the set of distance labels of the nodes of V.

PROPERTY 1 (Optimality Conditions). *Suppose that S denotes the set of source nodes and that t' denotes the current sink node. If there are no active nodes, then $(D, N - D)$ is a minimum capacity S–t' cut.*

PROPERTY 2 (W-Validity Property). *For each arc (i, j) in the residual network, if i and j are both in W, then $d(i) \leq d(j) + 1$.*

PROPERTY 3 (Increasing Distance Labels). *The distance labels are nondecreasing throughout the execution of the algorithm. Moreover, at the relabel of node i, either $d(i)$ is increased or else i is transferred from W to D.*

PROPERTY 4 (Dormancy Property). *The sink node t is in W. There is no arc of the residual network directed from a node in D to a node in W* (Extended Dormancy Property). *For each pair i, j of indices with $i < j$, there is no arc of the residual network directed from a node in* DormantSet(i) *to a node in* DormantSet(j).

PROPERTY 5 (d-Consecutiveness Properties). *$d(W)$ is a set of consecutive integers. Suppose that $R =$* DormantSet(j) *for some j. Then $d(R)$ is a set of consecutive integers.*

PROPERTY 6 (Bounds on Relabels). *Each node label is increased at most $n - 1$ times. In fact, $d(j) \leq n - 1$ for all nodes at all iterations.*

PROPERTY 7 (Bounds on Saturating Pushes). *Each arc (i, j) is saturated at most $n - 1$ times. The number of saturating pushes is $O(nm)$.*

PROPERTY 8 (Bounds on Transfers between W and D). *The number of times that a node or subset of nodes is transferred from D to W is less than n. The number of times that a node or subset of nodes is transferred from W to D is less than $2n$.*

Properties 1, 2, 3, and 7 are rather direct extensions of properties of the original G–T preflow push algorithm. We only outline their proofs here. The other properties are more specific to the algorithm presented here, and their proofs are presented in more detail.

Outline of Proof of Property 1. An S–t' preflow is called *maximum* if it maximizes the flow into the sink node t'. If x^* is a maximum S–t' preflow, then the flow into t' is the same as the maximum flow from S to t', which is the same as the minimum S–t' cut. (This is true since any preflow can be transformed into a flow without decreasing the flow into the sink node. See, for example, Goldberg and Tarjan [8].) Suppose that there are no active nodes, and that x' is the current preflow. Then the flow across $(D, N - D)$ (i.e., the sum of the flows of arcs directed from D to $N - D$

minus the sum of the flows of the arcs directed from $N - D$ to D) is the flow into t' since no node of $N - D - \{t'\}$ has any excess. It follows that the flow into node t' is also the capacity of the cut $(D, N - D)$, and thus this flow is maximum, and the cut is minimum. ■

Outline of the Proof of Property 2. The set of distance labels are W-valid at the beginning of the algorithm. It is easy to prove by induction that the distance labels remain W-valid following a push or relabel. (See, for example, Goldberg and Tarjan [8].) ■

Outline of the Proof of Property 3. Since the distance labels remain valid, at each iteration, there is never an arc (i, j) in the residual network with $d(i) > d(j) + 1$. Thus there is no need to reduce the distance label of a node. Moreover, the algorithm calls *ModifiedRelabel*(i) only when i has no admissible arcs. In this case, either i will have its distance label increased or a new DormantSet will be created, and i will be transferred into it, with other nodes possibly being transferred as well. ■

Proof of Property 4. This property is an extension of Lemma 3. We first note that the proof of Lemma 3 is valid for proving the dormancy property (but not the extended dormancy property) except in the case that nodes are transferred from D to W. However, it is easy to see that if the extended dormancy conditions are satisfied prior to transferring DormantSet($D_{\max}$) to W in a SelectNewSink procedure, then the dormancy conditions and the extended dormancy conditions are satisfied subsequent to the transfer. (We note that the significance of the extended dormancy conditions is that it permits the dormancy conditions to be satisfied subsequent to waking up DormantSet($D_{\max}$).

To complete the proof of property 4, we assume that the extended dormancy property is satisfied prior to some operation P, where P is not the transfer of nodes from D to W. It suffices to prove the extended dormancy property is still satisfied subsequent to carrying out operation P. If P is a push, the extended dormancy property remains satisfied. It also remains satisfied if P is a relabel of node i in which no node is transferred to D. Let us now consider the case in which *Modified Relabel*(i) is called, and i is transferred to D because there were no arcs from i to any other node in W. In this case, after the transfer, the algorithm sets DormantSet($D_{\max}$) to $\{i\}$. Suppose that $j' < j \leq D_{\max}$. If $j < D_{\max}$, then there was no arc directed from DormantSet(j') to DormantSet(j) prior to the operation P, and this remains true after P. Consider now the case in which $j = D_{\max}$. Prior to executing operation P, there was no arc directed from DormantSet(j') to a node in W, and thus subsequent to the operation, there is no arc directed from DormantSet(j') to DormantSet(j).

The remaining case is the one in which *Modified Relabel*(i) is called and i is the only node in W with distance label $d(i)$. Let $R = \{j \in W:$

$d(j) \geq d(i)\}$. The algorithm increments $D_{\max}$ by 1, and then lets DormantSet($D_{\max}$) = R. Suppose that $j' < j \leq D_{\max}$. If $j < D_{\max}$, then there was no arc directed from DormantSet(j') to DormantSet(j) prior to the operation P, and this remains true after P. Consider now the case in which $j = D_{\max}$. Prior to executing operation P, there was no arc directed from DormantSet(j') to a node in W, and thus subsequent to the operation, there is no arc directed from DormantSet(i) to DormantSet(j). ■

Proof of Property 5. Let us assume that the d-consecutiveness property holds at some iteration immediately prior to carrying out operation P. If P is a push, the d-consecutiveness properties are still satisfied subsequent to operation P since pushes do not affect distance labels. Consider next the case that P is a relabel of node i in which $d(i)$ is increased. Let $d'(\ )$ denote the distance labels subsequent to the relabel. The procedure *ModifiedRelabel*(i) would not have increased the distance label of node i unless there was another node $j \in W$ with $d(j) = d(i)$ prior to the relabel. It follows that $d(W) = d'(W - \{i\}) = d(W - \{i\})$, and thus $d'(W - \{i\})$ is a set of consecutive integers, say from k to k'. Then $d'(i) = d(j) + 1$ for some node $j \in W$, and $d'(W) = d(W - \{i\}) \cup \{d'(i)\}$. Since $k + 1 \leq d'(i) \leq k' + 1$, it follows that $d'(W)$ is a consecutive set of integers.

We now consider the case in which the procedure SelectNewSink is called, and $W = \varnothing$, and DormantSet($D_{\max}$) is transferred to W. Let W' denote the set of awake nodes subsequent to the transfer, and let R = DormantSet($D_{\max}$). By assumption, $d(R)$ was consecutive prior to the operation, and thus $d(W') = d(R)$ is consecutive.

We now consider the case in which *Modified Relabel*(i) is called, and nodes are transferred to D from W. Consider first the case in which there is some other node $j \in W$ with $d(j) = d(i)$ and there is no arc in the residual network directed from node i to another node in W. In this case, $D_{\max}$ is incremented, and DormantSet($D_{\max}$) is set equal to $\{i\}$. d(DormantSet($D_{\max}$)) = $d(i)$, and $d(i)$ is trivially consecutive. Moreover, after the operation, the new set of awake nodes is $W - \{i\}$, and $d(W - \{i\}) = d(W)$ by hypothesis. Thus $d(W - \{i\})$ is consecutive.

The last case that we consider is the case in which *ModifiedRelabel*(i) is called and there is no other node in W with distance label $d(i)$. In this case, we let $R = \{j \in W : d(j) \geq d(i)\}$. Since $d(W)$ is consecutive, it follows that $d(R)$ and $d(W - R)$ are both consecutive. Subsequent to this operation being executed, $D_{\max}$ is incremented, DormantSet($D_{\max}$) is set to be R, and W is replaced by $W - R$. It follows that the d-consecutiveness properties are satisfied in each of these cases. ■

Proof of Property 6. Here we will prove a slightly stronger property that will imply the results of Property 6. Let us define $d_{\min}(R) = \min(d(i) : i \in R)$, and let us define $d_{\max}(R) = \max(d(i) : i \in R)$. We claim that for each dormant set R that $d_{\min}(R) \le n - |R|$. We also claim that $d_{\min}(W) \le n - |W|$. Since $d(R)$ is a set of consecutive integers, these claims imply that $d_{\max}(R) \le d_{\min}(R) + |R| - 1 \le n - 1$. Similarly $d_{\max}(W) \le d_{\min}(W) + |W| - 1 \le n - 1$. Thus the claims together with the consecutiveness properties prove the validity of Property 6. We now prove that the claims are true.

The claims are easily seen to be valid after initialization when $|W| < n$ and $d(t') = 0$ for the first sink node t'. Assume that the claims are true prior to carrying out some operation P. The DormantSets do not change unless we are carrying out the operation *SelectNewSink* or unless we are transferring nodes from W to D. Thus, we can restrict attention to the case in which nodes are transferred from W to D or we are selecting a new sink.

Consider first the case that nodes are transferred from W to D in a relabel operation, and assume that the claim is true prior to the operation. The operation decreases the cardinality of W but does not modify $d_{\min}(W)$, and thus the property that $d_{\min}(W) \le n - |W|$ remains true subsequent to the operation. In the case that the only node transferred from W to D in *ModifiedRelabel*(i) is node i, the claims are still both true following the operation. We now consider the case that $R = \{j \in W: d(j) \ge d(i)\}$, and that $D_{\max}$ is incremented, and DormantSet$(D_{\max})$ is set equal to R. In this case, $d_{\min}(R) = d(i)$. Moreover, the set $d(W - R)$ is consecutive, and thus $d(i) \le d_{\min}(W) + |W - R| \le n - |W| + |W - R| = n - |R|$, and thus the claim is still true for set R.

We finally consider the selection of a new sink node. First of all, the old sink node t' transferred from W to DormantSet(0), and thus $|W|$ is decreased by 1. Simultaneously, if there are any other nodes in W, then there must be some node with distance label $d(t') + 1$, and thus $d_{\min}(W)$ is increased by at most 1. It follows that the claims remain true following a transfer of the old sink node to D. The claims also remain true if a dormant set is transferred to W or if a new sink node is selected. Thus in all cases, the claims remain valid. ∎

Proof of Property 7. As in the case of other distance based maximum flow algorithms, for each arc (i, j) there is at most one saturating push between consecutive relabels of node i. Since the number of distance increases for any node is less than n, it follows that the number of times that any arc (i, j) can be saturated is less than n, and the number of arc saturations in total is $O(nm)$. ∎

Proof of Property 8. The only time that a node is transferred from D to W is when $W = \varnothing$ and we are selecting a new sink node. This can happen at most $n - 1$ times. Thus the number of transfers from D to W is less than n. There are two types of transfers of nodes from W to D. The first type is the transfer of the old sink nodes, and this occurs at most $n - 1$ times. The second type leads the creation of new dormant sets; however each of these dormant sets is ultimately transferred back to W in a *SelectNewSink* procedure. Therefore this type of transfer from D to W can happen at most $n - 1$ times, completing the proof ■

5. Time Bounds for Different Implementations

In this section, we discuss the time bounds for different implementations of the minimum cut algorithm. We first briefly discuss data structures, and then we outline the different time bounds.

In selecting admissible arcs, we use the standard "current arc data structure." (See, for example, Ahuja *et al.* [1]. Briefly, the arcs emanating from node i are stored as a singly linked list $A(i)$. There is also a pointer called CurrentArc(i) that points to one of the arcs in $A(i)$. Initially, CurrentArc(i) points to the first arc of $A(i)$. When looking for an admissible arc in $A(i)$, one first checks to see if the arc pointed to by CurrentArc(i) is admissible. If it is not, then CurrentArc(i) is incremented until eventually it points at an admissible arc or else the arc list $A(i)$ is exhausted. In the latter case, one can show that there are no admissible arcs in $A(i)$, and then the procedure *ModifiedRelabel*(i) is called. Goldberg and Tarjan [8] showed that the time for searching for admissible arcs is dominated by the time spent in relabels plus the time spent in nonsaturating pushes.

For several different implementations, it is useful to maintain a set of lists Active(k) for each $k = 1$ to $n - 1$, where Active(k) denotes the set of active nodes whose distance label is k. These $n - 1$ sets are easily maintained at an additional cost of $O(1)$ per push and $O(1)$ per relabel of node i. In addition, for every node transferred from D to W or from W to D, there is an additional expense of $O(1)$ steps. This data structure is particularly useful when the node selection rule selects the active node with the largest distance label.

The *ModifiedRelabel*(i) procedure requires the knowledge of whether there is any other node in W with distance label $d(i)$. In order to implement this procedure, one can create an array denoted as numb(k) for $k = 1$ to $n - 1$, where numb(k) denotes the number of nodes in W whose distance label is k. This array is easily maintained at an additional cost of $O(1)$ steps per relabel of node i and a cost of $O(1)$ steps for every node transferred from D to W or from W to D.

The running time for the algorithm may be partitioned as follows: (1) the time for initializing, (2) the time for selecting active nodes, (3) the time for selecting admissible arcs, (4) the time spent in saturating pushes, (5) the time spent in nonsaturating pushes, (6) the time for increasing distance labels in relabel operations, (7) the time for creating dormant sets and thus for transferring nodes from W to D, and (8) the time spent in selecting a new sink node.

The time spent in initialization is $O(m)$. We will soon show that the time spent in selecting active nodes is $O(n^2 + \#$ of pushes), and the time spent in selecting admissible arcs is $O(nm + \#$ of pushes). The time spent in saturating pushes is $O(nm)$ by Property 7 on the bounds for saturating pushes. The time spent on increasing distance labels in a relabel operation is $O(nm)$ in total, as in the original G–T preflow push algorithm because each node has its distance label increased at most n times, and to increase the distance label of node i once takes $|A(i)|$ steps. The time spent in creating dormant sets is $O(n)$ per dormant set, and thus $O(n^2)$ in total, by Property 8. The time spent in selecting a new sink node is $O(1)$ per sink node plus the time spent in transferring nodes from D to W. This time is $O(n^2)$ in total since at most n sets of nodes are transferred from D to W, and each transfer takes $O(n)$ time. We now summarize these results.

THEOREM 4. *The procedure FindMinCuts*(s) *determines the minimum cut with s on the source side, and the running time is* $O(nm + \#$ *of non-saturating pushes*).

We have justified everything in Theorem 2 except for the time bound for selecting an active node. One natural rule for selecting an active node is to select the active node whose distance label is the highest. This is usually referred to as *highest level pushing*. We claim that this operation takes $O(n^2 + \#$ of pushes) to implement. We can see this as follows. At the beginning of each max flow problem, we can select the first active node by scanning the sets Active( ) in $O(n)$ time, for $O(n^2)$ steps in total. Next, suppose that at some iteration one has just pushed flow from node i. At the next iteration, assuming that the next operation is a push, one either pushes from a node with distance label $d(i)$ or else pushes from a node with distance label $d(i) - 1$. This selection can be carried out in $O(1)$ steps by scanning the sets Active($d(i)$) and Active($d(i) - 1$). The only time that the highest level of an active node can increase is following a distance increase of node i, at which point the highest level of an active node is the revised distance label for node i. Thus, selection in this case takes $O(1)$ steps.

The only case that we have not considered is in the *ModifiedRelabel* procedure when one transfers nodes from W to D. We have already seen that this can occur at most $2n$ times, so it is permissible to spend $O(n)$ steps per select without increasing the time bound beyond $O(n^2 + \#$ of pushes). This is easily accomplished by scanning the sets Active( ).

THEOREM 5. *The algorithm FindMinCut(s) with highest level pushing runs in $O(n^2m^{1/2})$ time.*

Proof. Cheriyan and Maheshwari [3] proved that the number of nonsaturating pushes of the highest level pushing algorithm is $O(n^2m^{1/2})$ when applied to the maximum s–t flow problem. The same proof technique directly extends to the algorithm *FindMinCut*(s) as well. (For an alternative proof, see Ahuja *et al.* [1].) ■

Another natural rule for selecting the active node is to store the set of active nodes as a queue. As per Goldberg and Tarjan [8], we refer to this rule as FIFO pushing. This rule is easily implemented in $O(1)$ time per push plus $O(1)$ time for each node transferred between D and W.

THEOREM 6. *The algorithm FindMinCut(s) with FIFO pushing runs in $O(n^3)$ steps.*

Proof. Goldberg and Tarjan proved that the FIFO rule leads to $O(n^3)$ nonsaturating pushes for the single source sink problem. Their potential function argument directly extends to the algorithm *FindMinCut*(s) as well. ■

THEOREM 7. *If dynamic trees are implemented in the same manner as in the G–T algorithm, then the algorithm FindMinCut(s) runs in $O(nm \log n^2/m)$ steps.*

Proof. The same proof given by Goldberg and Tarjan extends to the algorithm *FindMinCut*(s) as well. The only differences in the analysis involve the transfer of nodes between D and W, and these transfers are not the bottleneck operation. ■

6. BIPARTITE PROBLEMS

We now consider the problem of finding a minimum cut in a bipartite network $G = (N, A)$, where the node set N is partitioned into two parts N' and $N - N'$, where $|N'| = n'$. We assume without loss of generality that $n' \leq n - n'$.

LEMMA 8. *Let $G = (N, A)$ be a connected bipartite network and let N' be one of the parts of N. Then the minimum cut in G either is of the form $(\{i\}, N - \{i\})$, where $i \in N - N'$, or else there is a minimum cut $(S^*, N - S^*)$ where S^* contains a node of N'.*

Proof. We can assume that the nodes on each side of the cut are connected. If the smaller side of the cut has at least two nodes, then it has at least one node in N' and at least one node in $N - N'$, and this completes the proof. ■

Ahuja *et al.* [2] shows how to solve a bipartite maximum flow problem in time $O(n'm \log(2 + (n'^2/m)))$ steps using a bipush variant of the preflow push algorithm and using dynamic trees. The same approach extends to the algorithm *FindMinCut* as well; however, it needs a few minor modifications. First of all, as per Lemma 8, we can restrict the sink nodes to be nodes in N'. This means that $O(n')$ minimum $S - t'$ cut problems are solved. Property 8 can be modified accordingly, and the number of times that sets are transferred between D and W is $O(n')$. The time spent on carrying out these operations is $O(n'n)$. All other time bounds are as stated in the paper by Ahuja *et al.*, and we state this as the following theorem:

THEOREM 9. *Let $G = (N, A)$ be a connected bipartite network and let N' be one of the parts of N, and let $n' = |N'|$. Suppose that the algorithm FindMinCut(s) is implementing using bipushes and using the dynamic tree implementation of Ahuja et al. Then the resulting running time is* $O(n'm \log(2 + (n'^2/m)))$.

Proof. The same proof given by Ahuja *et al.* extends to the algorithm *FindMinCut*(s) as well. The only differences in the analysis involve the transfer of nodes between D and W, and these transfers are not the bottleneck operation.

7. FURTHER IMPLEMENTATION ISSUES

We conjecture that the algorithms of this paper will be quite effective in practice, and that their practical running time should be within a constant factor of the time to solve the first s–t cut problem. In fact, we have tailored the algorithm in some ways so that it will be more practical. For example, it is possible to prove that the algorithm would work even if all of the dormant sets were combined into a single set D, and D was awakened when W became exhausted. (Actually the proof that this works is not much easier than the proof that the current version of the algorithm works.) However, in practice one would prefer to have the set W be as small as possible, since a small set W leads to fewer pushes and relabels in general.

We now consider some additional data structures that should be more efficient in practice. We only mention a couple of data structures relating to manipulating the dormant sets, since other data structures can be adapted from good implementations of the preflow push algorithm. The algorithm needs to store each of the dormant sets as a singly linked list. This can be accomplished by maintaining an array of size n called DormantPointer and an array of size n called Successor. DormantPointer(i) gives the index of the node that is first in the set DormantSet(i). Suppose that j is a node in D, say that $j \in$ DormantSet(k). Then Successor(j) gives the index of the node following j on the list DormantSet(k), and if j is the last node on the list (or if j were in W) then Successor(j) $= 0$. In this way one can store all of the dormant sets quite efficiently.

In pushing flow from node i, one needs to be able to identify whether the arc (i, j) is admissible. For this, one needs to know whether $j \in W$. (It is not enough to check whether $d(i) = d(j) + 1$ since it is possible that this distance condition is true but that $y \in D$.) One obvious approach is to keep a bit-vector of n bits, where the jth bit is 1 or 0 according as j is or is not in W. A preferred approach is to increase $d(j)$ by n when j is transferred from W to D, and to decrease $d(j)$ by n when j is transferred from D to W. In this way, if $j \in D$ and if $i \in W$, then $d(j) > d(i)$, and arc (i, j) would be (correctly) labeled as inadmissible.

Consider a stage of the min cut algorithm when one wants to find the best S–t' cut, and suppose that the best current cut has capacity z^*. One other useful heuristic is to stop the procedure for finding the best S–t' cut as soon as the flow into node t' is at least z^*. At this point, the algorithm can select a new sink node. This heuristic can be added easily, and with almost no overhead.

We remark that one advantage of our approach is that we never need to explicitly contract nodes. In some sense, the algorithm implicitly contracts nodes. For example, when solving a maximum flow problem for source t', all of the nodes in D can be viewed as being contracted into a single node. In general, all of the nodes of S can be viewed as contracted into a single node. In each of these cases, there is no need to carry out the contraction, and there would be no gain in computational time from contracting these nodes.

8. Summary and Conclusions

We have presented an algorithm for finding the minimum unrestricted cut in either a directed or an undirected network, and whose running time is $O(nm \log n^2/m)$, which is the running time of the Goldberg–Tarjan algorithm for finding a maximum flow or minimum s–t cut.

The basic idea of our algorithm is simple. We solve $n - 1$ minimum S–t' cut problems, where each node in $N - \{s\}$ is a sink node for one of these $n - 1$ problems. After having found the minimum cut for sink node t', we transfer t' to S and select the node with minimum distance label. In general, one would expect that the distance label of the new sink would be equal to the distance label of the old sink node plus 1, and the algorithm proceeds using the old distance labels obtained by the algorithm at the end of solving the previous minimum cut problem. However, this procedure does not work in and of itself unless one is careful about relabeling nodes that have become disconnected from the sink. In our algorithm, we identified nodes disconnected from the sink in a very efficient manner, and we labeled these disconnected nodes as dormant. Much of the work in the algorithm and its analysis dealt with properties of these dormant nodes.

Acknowledgments

We acknowledge the help of George Kocur and Marge Hommel with an earlier draft of this manuscript. We also thank Charu Aggarwal for his editorial comments. We also acknowledge partial support through NSF Contract DDM-8921835 and through Air Force Office of Scientific Research Contract AFOSR-88-0088 ONR Contract N00014-94-0099.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network Flows: Theory, Algorithms, and Applications," Prentice–Hall, Englewood Cliffs, NJ, 1993.
2. R. K. Ahuja, J. B. Orlin, C. Stein, and R. E. Tarjan "Improved Algorithms for Bipartite Network Flows," Sloan School Working Paper 3218-90-MS, 1990.
3. J. Cheriyan and S. N. Maheshwari, "Analysis of Preflow Push Algorithms for Maximum Network Flow," Technical Report, Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi, 1987.
4. J. T. Cheriyan, T. Hagerup, and K. Mehlhorn, Can a maximum flow be computed in $O(nm)$ time? *in* "Proceedings of the 17th International Colloquium on Automata, Languages and Programming," pp. 235–248, 1990.
5. U. Derigs and W. Meier, "Implementing Goldberg's Max-Flow Algorithm: A Computational Investigation," Technical Report, University of Bayreuth, 1988.
6. L. R. Ford, Jr. and D. R. Fulkerson, Maximal flow through a network, *Canad. J. Math.* **8** (1956), 399–404.
7. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan, A fast parametric flow algorithm, *SIAM J. Comput.* **18** (1989), 30–55.
8. A. V. Goldberg and R. E. Tarjan, A new approach to the maximum flow problem, *J. Assoc. Comput. Mech.* **35** (1988), 931–940.
9. V. King, S. Rao, and R. E. Tarjan, A faster deterministic maximum flow algorithm, *J. Algorithms* **17** (1994), 447–474.
10. D. W. Matula, Determining edge connectivity in $O(nm)$, *in* "Proceedings of the 28th Annual Symposium on Foundations of Computer Science," pp. 249–251, 1987.
11. Y. Mansour and B. Schieber, "Finding the Edge Connectivity of Directed Graphs," Research Report RC 13556, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, 1988.

12. H. Nagamochi and T. Ibaraki, Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discrete Math.*, 1988.
13. H. Nagamochi and T. Ibaraki, Linear time algorithm for finding a sparse k-connected graph, *Algorithmica*, 1988.
14. M. Padberg and G. Rinaldi, An efficient algorithm for the minimum capacity cut problem, *Math. Programming* **47** (1990), 19–36.
15. J. C. Picard and M. Queyranne, Selected applications of minimum cuts in networks, *INFOR* **20** (1982), 394–422.