

Chapter 21

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

Jianxiu Hao*

James B. Orlin[#]

Abstract

We consider the problem of finding the minimum capacity cut in a 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 undirected network. The algorithm also determines the arc connectivity of either a directed or undirected network in $O(nm)$ steps.

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 [1956], 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 cut in a network in which no source or sink node is specified. The classic 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, Picard and Queyranne [1982] and Ahuja, Magnanti, and Orlin [1992] 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 non-empty 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 is no designated source and sink. Currently, the best algorithm for solving the minimum unrestricted cut problem (in terms of performance guarantees) is 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 is $O(nM(n,m))$, 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 for solving the maximum flow problem is an algorithm by King, Rao, and Tarjan [1991] and runs in $O(\min(nm + n^{2+\epsilon}, nm \log n))$ time where ϵ is any fixed positive constant.

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

* GTE Laboratories, Waltham, MA 02254

[#] MIT Sloan School of Management, Cambridge, MA 02139

⁺ This research was supported in part through NSF contract DDM-8921835 and through Air Force Office of Scientific Research contract AFOSR-88-0088.

solve all of the $O(n)$ minimum s - t cut problems in total time $O(nm \log n^2/m)$ steps. The algorithm relies on a simple idea, also exploited in the very efficient parametric flow algorithm developed by Gallo, Grigoriades, and Tarjan [1989]; i.e., 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.

When restricted to the problem of finding the arc-connectivity of the network, our algorithm runs in $O(nm)$ time, which is comparable to the best other algorithms (developed by Matula [1987] for the undirected problem and by Mansour and Schieber [1988] for the directed problem), under reasonable assumptions on the data.

The algorithm can also be sped up in the case that the network is bipartite, using the algorithm of Ahuja, Orlin, Stein and Tarjan [1990]. If the number of nodes on the smaller half of the bipartite network is $n' \ll n$, then the running time of the algorithm is $O(n'm \log((2 + n^2/m)))$.

In addition to having a good performance guarantee, we anticipate that our algorithm will be quite good in practice, and it is as easy to implement as any other preflow push algorithm for the maximum flow problem. In particular, it does not require contraction of nodes as does, for example, the algorithm of Padberg and Rinaldi [1990]. While contraction is conceptually quite simple, it is difficult to implement when one uses the usual network data structures such as a "forward star".

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, Magnanti and Orlin [1992]. We refer the reader to that text for further details not only on notation, but also for further details on the algorithm developed by Goldberg and Tarjan [1988].

Let $G = (N, A)$ be a 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 a associated capacity u_{ij} , which is a non-negative 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 non-empty 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 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 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. We now show how to find the minimum capacity cut with the property that s is on the source side of the cut. This algorithm, as presented in its most abstracted and high level, is essentially the same as the Padberg and Rinaldi algorithm. However, the

algorithm differs substantively at a more detailed level, and these differences account for our improved running times.

algorithm FindMinCut(s)

```

begin
  S := {s};
  BestValue := ∞;
  while S ≠ N do
    begin
      select some node t' ∈ 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
  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 minimum cut problem to the next. We first 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 side of the cut. \diamond

In this abbreviated version of our paper, we have omitted some of the proofs. These proofs are included in a full version of this paper.

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 we can solve these $n-1$ minimum S-t' cut problems in the time that it normally takes 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 [1988] algorithm for the maximum flow and minimum s-t cut problems. Using highest level pushing and without using dynamic trees, the running time is $O(n^2 m^{1/2})$, as is Cheriyan and Maheshwari's [1987] variant of the G-T maximum flow algorithm.

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 modification 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 source nodes and t is a designated sink. 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 single 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 property that there is never an arc of the "residual network" directed from a dormant node to a node that is awake.

In Section 4, we will show how to modify the preflow-push algorithm of this section so as 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 function on the arcs such that $0 \leq x_{ij} \leq u_{ij}$ 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:

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

We say that x is a *preflow* if $0 \leq x_{ij} \leq u_{ij}$ for each $(i,j) \in A$ and if $e(i) \geq 0$ 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 $(i,j) \in G$, 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 algorithm, 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 $d(i) \leq d(j) + 1$.

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

As a result, if 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 [1988], and is easily proved via induction.

Lemma 2. Suppose that the set $d(\cdot)$ 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 $G(x)$. \diamond

A node i is 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 if $r_{ij} > 0$. In general, the

algorithm will send flow from active nodes and will send flow along admissible arcs. When a node i is active but there are no admissible arcs emanating from node i , then node i will be relabeled.

In the procedure *Relabel(i)*, the algorithm will either set the distance label of node i to $\min(1 + d(j) : r_{ij} > 0 \text{ and } j \in W)$ or else it will transfer node i and possibly other nodes from W to the set D of dormant nodes. The first operation is the standard relabel operation of the Goldberg-Tarjan preflow push algorithm. The transfer of node i and possibly other nodes to D distinguishes our implementation from theirs. Furthermore, we will partition the set D into subsets called $DormantSet(0)$, $DormantSet(1)$, \dots , $DormantSet(D_{max})$ 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 $D = \bigcup_{i=1}^K DormantSet(i)$. 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 will be the set S of source nodes. The rationale for having the set D or dormant nodes and for further partitioning set D into subsets $DormantSet(i)$ for $i = 1$ to D_{max} will be discussed in the next section.

We are now ready to give the preflow push algorithm. We let S designate the set of source nodes and we let t' designate the sink node.

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 node  $s \in S$  and for each arc  $(s,j)$  with  $j \notin S$ ,
    send  $r_{sj}$  units of flow in  $(s,j)$ ;
  DormantSet(0) := S;
   $D_{\max} := 0$ ;
   $W := N - S$ ;
   $d(t') := 0$ ;
  for each  $j \in N - \{t'\}$  do  $d(j) := 1$ ;
end

procedure Relabel(i);
begin
  if  $d(j) \neq d(i)$  for each  $j \in W - \{i\}$  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  $d(i) := \min \{d(j)+1 : (i,j) \in A(i), j \in W \text{ and } r_{ij} > 0\}$ ;
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 $W = N - D$, and the set D is further partitioned into subsets DormantSet(0), . . . , DormantSet(D_{\max}). 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 . In fact, the algorithm maintains the following further property:

Extended Dormancy Property. We say that nodes in $D = \cup_k \text{DormantSet}(k)$ satisfy the *extended dormancy property* if there is no arc in the residual network directed from a node in DormantSet(i) to a node in DormantSet(j) for $i < j$.

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 DormantSet(0) as the set S , and it initializes 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 increments D_{\max} , and it transfers the nodes of R from W to DormantSet(D_{\max}). 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 dormant sets will still satisfy both the dormancy property and the extended dormancy property, assuming that D satisfied these two dormancy property prior to the transfer of the set R from W to D .

Lemma 3. Suppose that the set $d(\cdot)$ of distance labels is W -valid and that the set $D = N - W$ satisfies the dormancy and extended dormancy properties. 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, after the relabel the set D and W still satisfy the dormancy and extended dormancy properties. \diamond

We note that the procedure for identifying dormant nodes was presented as a heuristic for speeding up the computations in Derigs and Meier [1988]. It is also presented in Ahuja, Magnanti and Orlin [1992]. 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 subtract it from W . It is easy to see that the resulting set $D \cup \{i\}$ satisfies the dormancy property so long as D satisfied 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. Moreover, in Section 4, we will outline the proof of 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
  S := {1}; t' := {2};
  Initialize
  while S ≠ 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 Relabel(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
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 outline the proof of the correctness of the algorithm.

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 j in W with minimum distance label, and so long as there is such a node $j \in W$, it will be true that $d(j) = d(t') + 1$; 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 $\text{DormantSet}(D_{\max})$ to W , which makes W non-empty again. Once W is non-empty, the algorithm selects the node in W with minimum distance label, and starts a new minimum cut procedure.

We now see why we can refer to the nodes in D as dormant. Each of the nodes in D (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 *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 ∈ N-S do send rt'k units of
  flow in arc (t',k);
  if W = ∅ then
    begin
      W := DormantSet(Dmax);
      Dmax := Dmax - 1;
    end
  select j ∈ 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 outline their proof subsequently. For a given set V of nodes, we let $d(V) = \{k : d(j) = k \text{ for some } j \in V\}$. 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 non-decreasing 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). 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 2, 3, 4 and 5 can be proved directly by induction on the number of operations performed. One assumes that the property is satisfied prior to carrying out an operation P and proves that it remains satisfied after performing P . We now outline the proofs of the other properties.

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 by the max flow min cut theorem since any preflow can be transformed into a flow without decreasing the flow into the sink node. See, for example, Goldberg and Tarjan [1988].) 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. \diamond

Outline of the 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 $d_{\min}(R) \leq n - |R|$. We also claim that $d_{\min}(W) \leq n - |W|$. Since $d(R)$ is a set of consecutive integers, these claims imply that $d_{\max}(R) \leq d_{\min}(R) + |R| - 1 \leq n-1$. Similarly $d_{\max}(W) \leq d_{\min}(W) + |W| - 1 \leq n-1$. Thus the claims together with the consecutiveness properties prove the validity of Property 6. As with the Properties 2-5, the above claim can be proved using induction on the number of operations performed. \diamond

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 saturations in total is $O(nm)$. \diamond

Proof of Property 8. The only time that a node is transferred from D to W is when $W = \emptyset$ 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. \diamond

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 "current arc data structure" as described in Goldberg and Tarjan [1988]. Goldberg and Tarjan showed that the time for searching for admissible arcs is dominated by the time spent in relabels plus the time spent in non-saturating 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 $Relabel(i)$ 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 $O(1)$ 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.

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 non-saturating pushes, (6) the time for increasing distance labels in relabel operations, (7) the time for creating dormant sets, and thus 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 $O(n^2 + \# \text{ of pushes})$, and the time spent in selecting admissible arcs is $O(nm + \# \text{ of pushes})$. 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. Suppose that the procedure `FindMinCut(s)` is implemented so that the time to select an active node is $O(1)$ steps per push. Then the procedure `FindMinCut(s)` determines the minimum cut with s on the source side, and the running time is $O(nm + \# \text{ of non-saturating pushes})$.

Proof. The only case that we have not already considered is in the `Relabel` 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 $O(n^2 + \# \text{ of pushes})$. This is easily accomplished by scanning the sets $Active(\cdot)$. \diamond

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

Proof. Cheriyan and Maheshwari [1987] proved that the number of non-saturating pushes of the highest level pushing algorithm is $O(n^2 m^{1/2})$ 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, Magnanti and Orlin [1992]).

Another natural rule for selecting the active node is to store the set of active nodes as a queue. As per Goldberg and Tarjan [1992], we also call this rule 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 $O(n^3)$ non-saturating pushes for the single source single sink problem. Their potential function argument directly extends to the algorithm FindMinCut(s) as well. \diamond

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. \diamond

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. \diamond

Ahuja, Orlin, Stein and Tarjan [1990] show 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, Orlin, Stein, and Tarjan. Then the resulting running time is $O(n'm \log (2 + (n'^2/m)))$.

Proof. The same proof given by Ahuja, Orlin, Stein, 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. \diamond

Determining Arc Connectivity

The *arc connectivity problem* is the problem of finding the minimum number of arcs whose deletion disconnects the network. This problem is really a special case of the minimum cut problem in which each arc has a capacity of 1. Since every push is saturating, the number of non-saturating pushes is 0. We are thus led to the following conclusion.

Theorem 10. The algorithm FindMinCut(s) determines the arc connectivity of a network in $O(nm)$ steps. \diamond

The best other algorithms for determining the arc connectivity of a network are as follows: Matula [1987] gave an $O(nm)$ algorithm for finding the arc connectivity in an undirected network. If we let the arc connectivity be denoted as λ , Matula showed that his algorithm can be improved to run in $O(\lambda n^2)$ time, which dominates the $O(nm)$ bound when λ is asymptotically smaller than m/n . Mansour and Schieber [1988] developed an $O(nm + \lambda^2 n^2)$ algorithm for solving the arc connectivity problem on directed graphs. Our algorithm achieves the same running time if $\lambda^2 \geq m/n$.

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. In addition, our algorithm is comparable to the best algorithm for finding the arc connectivity or the node connectivity of a network under the plausible assumption that the arc connectivity (or node connectivity) is proportional to the average degree of the network.

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 is 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.

Acknowledgment

We like to acknowledge George Kocur and Marge Hommel for help in an earlier draft of this manuscript.

References.

- Ahuja, R. K., T. L. Magnanti, and J. B. Orlin. 1992. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, N. J.
- Ahuja, R. K., J. B. Orlin, C. Stein, and R. E. Tarjan. 1990. Improved Algorithms for Bipartite Network Flows. Accepted for publication by *Mathematical Programming*.
- Cheriyian, J. and S. N. Maheshwari. 1987. Analysis of Preflow Push Algorithms for Maximum Network Flow. Technical report, Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India.
- Derigs, U., and W. Meier. 1988. Implementing Goldberg's Max-Flow Algorithm: A Computational Investigation. Technical Report, University of Bayreuth, West Germany.
- Ford, L.R., Jr., and D. R. Fulkerson. 1956. Maximal Flow Through a Network. *Canadian Journal Math.* 8, 399-404.
- Gallo, G., M. D. Grigoriadis, and R. E. Tarjan. 1989. A Fast Parametric Flow Algorithm. *SIAM Journal of Computing*, 18, 30-55.
- Goldberg, A.V., and R.E. Tarjan. 1988. A New Approach to the Maximum Flow Problem. *Journal of the ACM* 35, 921-940.
- King V, S. Rao, and R. E. Tarjan. 1991. A Faster Deterministic Max-flow Algorithm. Manuscript in preparation.
- Matula, D. W. 1987. Determining Edge Connectivity in $O(nm)$. *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. 249-251.
- Mansour, Y and B. Schieber. 1988. Finding the Edge Connectivity of Directed Graphs. Research Report RC 13556, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, N.Y.
- Padberg, M. and G. Rinaldi. 1990. An efficient algorithm for the minimum capacity cut problem. *Math. Programming* 47, 19-36.
- Picard, J. C. and M. Queyranne. 1982. Selected applications of minimumcuts in networks. *INFOR* 20, 394-422.