[3] M. L. DOWLING, *A mathematical theory for code parallelisation*, Ph.D. thesis, Carolo-Wilhelmina Universität zu Braunschweig, Braunschung, FRG, 1987.
[4] P. HENRICI, *Applied and Computational Complex Analysis, Vol.* 1, John Wiley, New York, 1974.
[5] W. G. HORNER, *Philosophical Transactions of the London Mathematical Society*, 109 (1819), pp. 308–335.
[6] L. HYAFIL, *On the parallel evaluation of multivariate polynomials*, SIAM J. Comput., 8 (1979), pp. 120–123.
[7] D. E. KNUTH, *The Art of Computer Programming, Vol.* 2: *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.
[8] D. KERSHAW, *Solution of single, tridiagonal systems and vectorisation of the* ICCG-*Algorithm on the* Cray-1, in Parallel Computations, G. Rodrigue, ed., Academic Press, New York, 1982.
[9] L. KRONSJÖ, *Algorithms: Their Complexity and Efficiency*, 2nd ed., John Wiley, New York, 1987.
[10] D. J. KUCK, R. H. KUHN, B. LEASURE, AND M. WOLFE, *Advanced, retargetable vectoriser*, IEEE Tutorial for Super-Computers: Design and Applications, K. Hwang, ed., 1984, pp. 186–203.
[11] L. LAMPORT, *The parallel execution of* DO-*loops*, Comm. ACM, 17 (1974), pp. 83–93.
[12] W. RÖNSCH, *Stability aspects in using parallel algorithms*, Parallel Comput., 1 (1984), pp. 75–98.
[13] G. RODRIGUE, N. MADSEN, AND J. KARUSH, *Odd-even reduction for banded linear equations*, J. Assoc. Comput. Mach., 26 (1979), pp. 72–81.
[14] L. G. VALIANT, S. SKYUM, S. BERKOWITZ, AND C. RACKOFF, *Fast parallel computation of polynomials using few processors*, SIAM J. Comput., 12 (1983), pp. 641–644.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 143–155, February 1990

VERY SIMPLE METHODS FOR ALL PAIRS NETWORK FLOW ANALYSIS*

DAN GUSFIELD†

Abstract. A very simple algorithm for the classical problem of computing the maximum network flow value between every pair of nodes in an undirected, capacitated n node graph is presented; as in the well-known Gomory-Hu method, the method given here uses only $n-1$ maximum flow computations. Our algorithm is implemented by adding only five simple lines of code to any program that produces a minimum cut; a program to produce an *equivalent flow tree*, which is a compact representation of the flow values, is obtained by adding only three simple lines of code to any program producing a minimum cut. A very simple version of the Gomory-Hu *cut tree* method that finds one minimum cut for every pair of nodes is also derived, and it is shown that the seemingly fundamental operation of that method, node contraction, is not needed, nor must crossing cuts be avoided. As a result, this version of the Gomory-Hu method is implemented by adding less than ten simple lines of code to any program that produces a minimum cut. The algorithms in this paper demonstrate that a cut tree of graph G can be computed with $n-1$ calls to an oracle that alone knows G, and that, when given two nodes s and t, returns any arbitrary minimum (s, t) cut and its value.

Key words. network flow, combinatorial optimization

AMS(MOS) subject classifications. 90B10, 90B35, 90C35, 68Q25, 05C99

1. Introduction. For an undirected graph G with n nodes, Gomory and Hu [GH] showed that the flow values between each of the $n(n-1)/2$ pairs of nodes can be computed by solving only $n-1$ network flow problems on G, saving a factor of n over the obvious method. Furthermore, they showed that the flow values can be represented by a weighted tree T on n nodes, where for any pair of nodes (x, y), if e is the minimum weight edge on the path from x to y in T, then the maximum flow value from x to y in G is exactly the weight of e. Such a tree is called an *equivalent flow* tree. They also showed a stronger result, that there exists an equivalent flow tree, where for every pair of nodes (x, y), if e is as above, then the two components of $T-e$ form a minimum cut between x and y in G. Such a tree is called a *GH cut tree*, and it compactly represents one minimum cut for each pair of nodes. Figure 1 shows a three node graph G, a cut tree T of G, and an equivalent flow tree T' of G. Note that T' is not a cut tree of G. The method given in [GH] produces a GH cut tree using only $n-1$ maximum flow computations. This method is well known and is discussed in many texts and surveys on graphs and network flows [H1], [H2], [LP], [FF], [FR, FR], [LP], [HA], [PG], [VL], as well as in technical papers which build on it [AMS], [AH], [E], [H3],

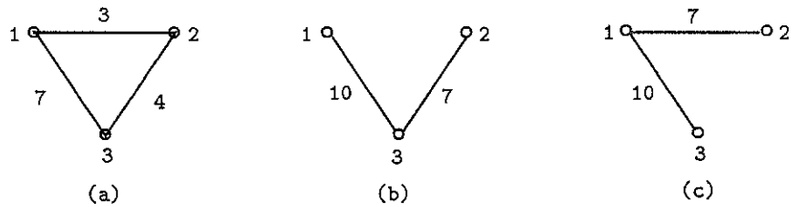

FIG. 1. *Graph G, a cut tree T, and an equivalent flow tree T'.*

* Received by the editors August 17, 1987; accepted for publication (in revised form) April 7, 1989. This research was partially supported by U.S. Census Bureau grant JSA 86-9 and National Science Foundation grant CCR-880374.

† Computer Science Division, University of California at Davis, Davis, California 95616.

[HR], [HS], [SC], [S], [T], [GrH]. For a basic discussion of graphs and network flows, see [FF], [L], or [H2]. For a textbook discussion of the GH method, see [H2] or [FF].

Two cuts (X, Y) and (U, V) are said to *cross* if all four set intersections, $X \cap U$, $X \cap V$, $Y \cap U$, and $Y \cap V$, are nonempty. The Gomory-Hu method, and methods based on it, require that all the cuts computed be pairwise noncrossing. Most of the work of the method, other than the work involved in the maximum flow computations, is involved in explicitly maintaining the noncrossing condition, or is a consequence of that condition. In particular, the operations of node contraction and identification of which nodes to contract, are consequences of the need to maintain noncrossing cuts. In all discussions of the GH method that we know of, both algorithmic and mathematical, the existence of noncrossing cuts has been fundamental to both the logic of cut trees, and to the algorithms to find and use them.

The GH method is fairly involved and nontrivial to program. A different method for computing all the flow values, and a cut tree, can be obtained by modifying a method of Schnorr [SC] for a related problem on directed graphs. This method requires $O(n \log n)$ maximum flow computations, but it can be implemented to have an amortized total running time of $O(n^4)$. However, the implementation is more complex than the GH method, and to obtain the faster time bound, or to build cut trees, the method also needs to maintain noncrossing (directed) cuts.

As for equivalent flow trees, in most of the published literature a full GH cut tree is used even when only the flow values are required. However, after the results in this paper were first obtained [GU1], we learned of a related method by Granot and Hassin [GrH] which can easily be modified to produce an equivalent flow tree, but not a cut tree. That method solves only $n-1$ maximum flow problems, and does not need to maintain noncrossing cuts. Hence, that is the first paper we know of that indicated that crossing cuts can be used in computing equivalent flow trees.

In this paper we give simple, efficient methods which show that crossing cuts can be used in producing GH cut trees as well as equivalent flow trees. We first give an extremely simple, efficient algorithm for producing an equivalent flow tree that is not necessarily a cut tree; as in the GH method, only $n-1$ maximum flows are computed by the method. The simplicity of the method comes from the fact that the method does not need to avoid crossing cuts, and so does not need to contract nodes. We implement the method by adding only three simple lines of code to any maximum flow program that produces a minimum cut; the program can be extended to explicitly output the $n(n-1)/2$ flow values, by adding only two additional lines of code. We next show that with a modification of the Gomory-Hu cut tree method, noncrossing cuts need not be maintained, and so the fundamental operation of node contraction is not needed, and the intermediate cut trees need not be explicitly represented or searched. Hence, the major programming and data structures details needed for the original GH method can be avoided. As a result, any maximum flow program producing a minimum cut can be converted to one that efficiently computes a GH cut tree, with the addition of under ten simple lines of code. More generally, we show that noncrossing cuts, which are central to all previous expositions on cut trees, are never explicitly needed in efficient algorithms for finding either cut trees or equivalent flow trees.

2. Equivalent flow trees and all pairs maximum flow.

ALGORITHM EQ. Input to the algorithm is an undirected capacitated graph G; output is an equivalent flow tree T'. The algorithm assumes the ability to find a minimum cut between two specified nodes in G.

1. Create a (star) tree T' on n nodes, with node 1 at the center and nodes 2 through n at the leaves.
2. For s from 2 to n do steps 3 and 4.
3. Compute a minimum cut (X, Y) in G between (leaf) node s and its (unique) neighbor t in T'. Label the edge (s, t) in T' with the capacity of (X, Y).
4. For every node i *larger than* s, if i is a neighbor of t, and i is on the s side of (X, Y), then modify T' by disconnecting i from t, and connecting i to s. Note that each node i larger than s remains a leaf in T'.

It is easy to see that at every iteration, node s and all nodes larger than s are leaves in T', so each chosen s has a unique neighbor, as expected by the algorithm. Figure 2 gives an example of the algorithm. Figure 2(a) shows the graph G, and the five cuts used by the algorithm; the capacity on each edge in G is one. Figure 2(b) shows tree T' before any cuts are computed; Figure 2(c) shows the tree after the first cut (1, 2) is computed; Figure 2(d) shows the final equivalent flow tree for G. Note that in this example the (5, 1) and the (3, 1) cuts each cross the (1, 2) cut. Also note that the equivalent flow tree T' of Fig. 1 would be obtained from running Algorithm EQ on the graph G of Fig. 1, illustrating the fact that Algorithm EQ does *not* always produce a cut tree.

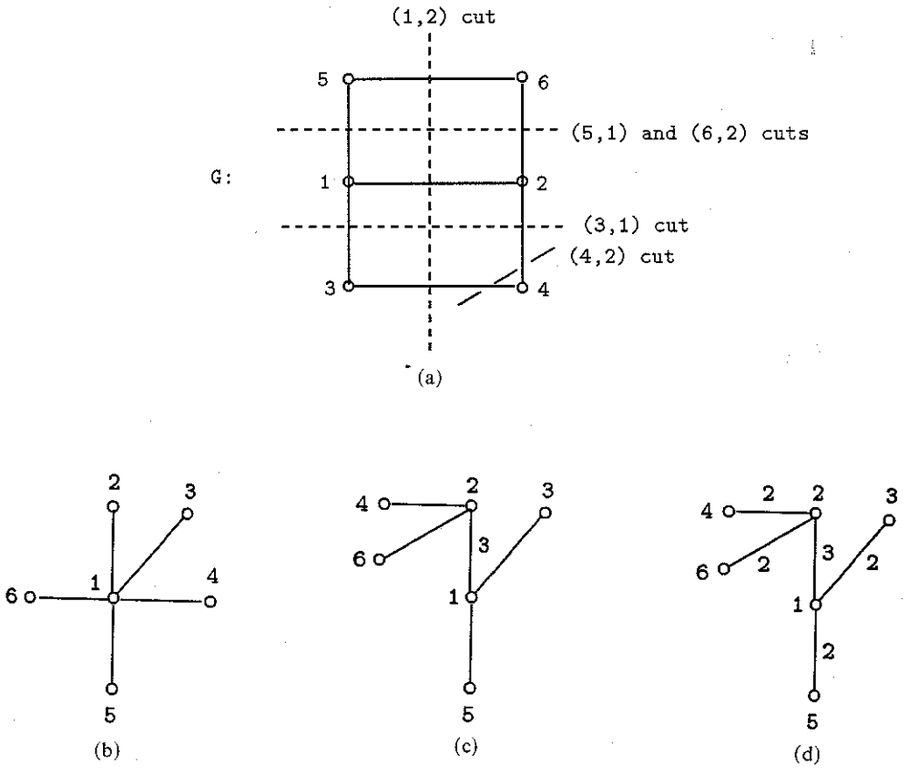

FIG. 2. *Graph G, and the creation of equivalent flow tree T' for G.*

To show the extreme simplicity of this method, we present the following "program" which implements Algorithm EQ. In the program, p is an n length vector initialized to 1; at every iteration, every node i larger than or equal to s is a leaf, and $p[i]$ indicates its unique neighbor. The program takes in graph G and outputs a set of weighted edges which form an equivalent flow tree T' of G.

PROGRAM EQ.

```
for s:=2 to n do
 begin
  Compute a minimum cut between nodes s and t:=p[s] in G;
  let X be the set of nodes on the s side of the cut.
  Output the edge (s, t) and the maximum s, t flow value f(s, t).
 for i:= s to n do
  if (i is in X and p[i]=t) then p[i]:=s;
 end;
```

To produce all the $n(n-1)/2$ flow values, let F be an n-by-n array, initialized to infinity, holding the flow values. Then insert the following lines before the "end;" above.

```
F[s, t]:=F[t, s]:=f(s, t);
for i:=1 to s-1 do
 if (i <> t) then F[s, i]:=F[i, s]:=min(f(s, t), F[t, i]);
```

In addition to the simplicity of the algorithm, it is noteworthy that the only interaction with graph G occurs inside the minimum cut routine. Hence, the algorithm can be thought of as $n-1$ calls to an oracle which alone knows the structure of G. Furthermore, for any given pair (s, t), if there is more than one minimum s-t cut, then the oracle (or adversary) is free to choose one arbitrarily. Thus, an equivalent flow tree for an unknown graph can be inferred from $n-1$ cut queries. We shall see that this is true for the cut tree as well.

We will present below a short, direct proof of the correctness of Algorithm EQ. A different, indirect, proof based on comparing the behavior of Algorithm EQ with the GH method is given in [GU1]. Before presenting the direct proof, we state some needed results initially shown in [GH].

LEMMA 1 [GH].[1] *Let (X, Y) be a minimum cut in G separating nodes $x \in X$ and $y \in Y$. Let u and v be two nodes on the X side of the cut, and let (U, V) be an arbitrary minimum (u, v) cut in G. If $y \in U$, then $(U', V') = (U \cup Y, V \cap X)$ is a minimum (u, v) cut, else (when $y \in V$) $(U', V') = (U \cap X, V \cup Y)$ is a minimum (u, v) cut.*

Figure 3 shows the two possibilities described by Lemma 1; cuts (X, Y) and (U, V) are drawn with straight solid lines, and cut (U', V') is drawn with a right angle, and marked by hatch marks. Note that in Lemma 1, it does not matter whether x is in U or in V; in Fig. 3 we have drawn x to be in U.

The importance of Lemma 1 is that it proves there always exists a minimum (u, v) cut (U', V') in G such that Y falls entirely on the u side or entirely on the v side of (U', V'). Hence (U', V') does not cross (X, Y). The existence of a noncrossing cut (U', V') is all that is needed in the correctness proof of the original GH method, but in this paper we use the following immediate, but key, corollary.

[1] The original lemma in [GH] is somewhat weaker, but the statement given here is explicitly stated and proved in the body of the proof of the original version. For the easiest such proof of Lemma 1, see [FF, p. 179] or [H2, pp. 66-68].

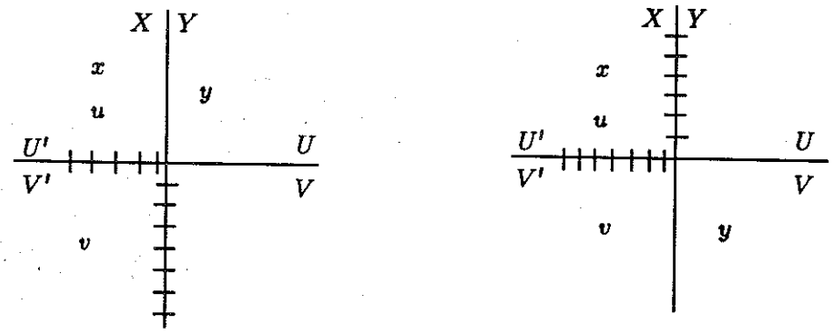

FIG. 3. *The two cases of Lemma* 1.

COROLLARY 1. *Let (X, Y), (U, V), and (U', V') be as in Lemma* 1. *Then the minimum (u, v) cut (U', V') does not cross (X, Y), and it splits X exactly the same way that (U, V) does.*

The following two facts are shown in [GH] (also in [FF] and [H2]) and are simple to prove.

LEMMA 2 [GH]. *Let $f(x, y)$ denote the maximum flow value between nodes x and y. If $\{v_1, v_2, \cdots, v_k\}$ is a set of nodes in G, then $f(v_1, v_k) \geqq \min [f(v_i, v_{i+1}): i = 1 \text{ to } k-1]$.*

COROLLARY 2 [GH]. *If i, j, and k are three arbitrary nodes in G, then the minimum of $f(i, j)$, $f(i, k)$, and $f(j, k)$ is not unique.*

2.1. Correctness of Algorithm EQ. Consider each edge (s, t) created in step 3 of the algorithm to be directed from s to t; then all edges are directed from larger node label to smaller node label, and hence T' is a directed tree where every directed path leads to node 1. For any path P (directed or not), let $\min(P)$ be the minimum weight of the edges on P.

LEMMA 3. *Suppose node i reaches node j by a directed path $P[i, j]$ in the final T', and suppose that (k, j) is a directed edge into j, where k is smaller (has smaller label) than any node on $P[i, j]$ except j. Then node i was a neighbor of j in T' at the time when the (k, j) cut C was computed by Algorithm* EQ. *Furthermore, i is on the k side of C if and only if k is on the directed path $P[i, j]$ in the final T'.*

Proof. At the start of the algorithm, node i is a neighbor of node 1 only. Then until iteration $i-1$, when i is node s in step 2 of the algorithm, node i has exactly one neighbor at any time, and the unique neighbor of i can change from v to w only when v is t and w is s in step 2. Hence every node on $P[i, 1]$ is a neighbor of i at some point before iteration $i-1$, and no node not on $P[i, 1]$ is. Then since $j < k$, j must be i's neighbor before the (j, k) cut C was computed. Furthermore, since k is smaller than every node on $P[i, j]$ except j, j must be the neighbor of i when C is computed. Now if k is on $P[i, j]$, then i surely is on the k side of C, and if k is not, then i cannot be on the k side. □

THEOREM 1. *Given input graph G, Algorithm* EQ *correctly computes an equivalent flow tree T' for G.*

Proof. First, note that if (x, y) is an edge in T', then Algorithm EQ computed an (x, y) minimum cut, and its value is written on edge (x, y). Hence the tree is correct for every pair of neighboring nodes in T'. Now we show that if (x, y) is an arbitrary pair of nodes not connected by an edge in T', and $P[x, y] = \{x = v_1, \cdots, v_k = y\}$ is the path (ignoring edge directions) in T' from x to y, then $f(x, y) = \min [f(v_i, v_{i+1}): i = 1 \text{ to } k-1]$. Given Lemma 2, we need only to show that $f(x, y) \leqq \min [f(v_i, v_{i+1}): i = 1$

to $k-1$]. Suppose not, and let (x, y) be the pair with shortest path $P[x, y]$ among all pairs where $f(x, y) > \min(P[x, y])$.

Case 1. Path $P[x, y]$ is a directed path from x to y (the case when it is directed from y to x is identical). Let $v \neq x$ be the neighbor of y on $P[x, y]$ (if $x = v$, the edge (x, y) is in T'). By the minimality of $P[x, y]$, $f(x, v) = \min(P[x, v])$, and since $f(x, y)$ is assumed to be greater than $\min(P[x, y])$, Corollary 2 implies that $\min(P[x, y]) = f(x, v) = f(v, y)$. But by Lemma 3, the cut between nodes y and v found by Algorithm EQ separates x and y, so $f(x, y) \leqq f(v, y) = \min(P[x, y])$, a contradiction.

Case 2. Path $P[x, y]$ consists of two directed subpaths $P[y, z]$ and $P[x, z]$, where $P[y, z]$ is directed from y to z and $P[x, z]$ is directed from x to z. Node z can be thought of as the least common ancestor of x and y in T' when node 1 is the root. Let x_1 be the neighbor of z on $P[x, z]$ and let y_1 be the neighbor of z on $P[y, z]$. Assume that $x_1 < y_1$, so in the running of Algorithm EQ the (x_1, z) cut, $C(x_1, z)$, was computed before the (y_1, z) cut.

From Case 1 we know that $f(x, z) = \min(P[x, z])$ and $f(y, z) = \min(P[y, z])$, so either $f(x, z)$ or $f(y, z)$ equals $\min(P[x, y])$. Hence by the assumption that $f(x, y) > \min(P[x, y])$, Corollary 2 says that $f(x, z) = f(y, z) = \min(P[x, y])$, and so there is an edge of weight $\min(P[x, y])$ on path $P[x, z]$. Let $e = (u, v)$ be the edge closest to z on $P[x, z]$ with weight $\min(P[x, y])$, let $C(u, v)$ be the (u, v) cut of that weight found by EQ, and let v be closer to z on $P[x, z]$ than u is. Then by Lemma 3, x, u, and v fall on the x_1 side of the cut $C(x_1, z)$ computed by the Algorithm EQ, and y falls on the z side of $C(x_1, z)$. By Lemma 3 again, x falls on the u side of $C(u, v)$, and from the assumption that $f(x, y) > \min(P[x, y])$, y must also fall on the u side. Figure 4 shows the general situation. In particular, the positions of nodes u, v, x, and y are each determined down to one of the four quadrants defined by the intersections of $C(x_1, z)$ and $C(u, v)$; the positions of nodes x_1 and z are each determined only to two quadrants.

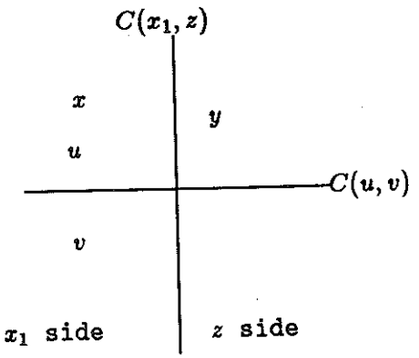

FIG. 4. *Case* 2 *of the proof of Theorem* 1.

Now there are two cases for the position of z. In either case, Lemma 1 can be applied (recall that in Lemma 1 the only assumption on the position of x_1 is that it is in X), yielding a minimum (u, v) cut C^* that either separates x and y, or that separates z and v. In particular, if $z \in U$, then the quadrant containing v defines a minimum (u, v) cut, and this cut also separates v and z; if $z \in V$, then the quadrant containing u defines a minimum (u, v) cut that also separates x from y. But, the minimum (u, v) cut has capacity $\min(P[x, y])$, so if C^* separates x and y, then $f(x, y) \leqq \min(P[x, y])$,

and so $f(x, y) = \min(P[x, y])$ as claimed. If C^* separates v and z, then $f(v, z) \leqq \min(P[x, y])$. But $P[v, z]$ is a directed path in T', so from Case 1, $f(v, z) = \min(P[v, z])$ and $\min(P[v, z]) > \min(P[x, y])$ by the selection of v, so $f(v, z) > \min(P[x, y])$. This gives a contradiction, and we conclude that $f(x, y) \leqq \min(P[x, y])$, so $f(x, y) = \min(P[x, y])$, and the correctness of Algorithm EQ is proved. □

3. A simple algorithm for the GH cut tree. In this section we show how to modify the GH method to avoid node contraction and the maintenance of noncrossing cuts. The result is a very simple algorithm to find a GH cut tree. The key idea is to show that although the original GH method must find in each step a minimum (u, v) cut that does not cross any previously used cuts, a modification of the method permits *any* minimum (u, v) cut to be used. The modified method will be proved correct by showing how its execution simulates a possible execution of the original GH algorithm.

DEFINITION. For a subset N_i of nodes of G, the *contraction* of N_i is the replacement of the nodes of N_i by a single node c_i, and for each node $v \in G - N_i$, the replacement of the edges from v to N_i with a single edge from v to c_i; the capacity of edge (v, c_i) is the sum of the capacities of the removed edges incident with v.

3.1. The Gomory–Hu method.

Input: An n node capacitated undirected graph G.
Output: A GH cut tree T for G.

1. Set T to be a single "supernode" containing every node of G. Then iterate the following step until every supernode contains only one node of G.
2. Pick a supernode S containing more than one node of G, and pick two nodes u and v in S. Find all the connected components of $T - S$ and let N_i be the set of nodes of G contained in the supernodes of the ith connected component of $T - S$. Successively contract the nodes in each set N_i in G, and let $G(S)$ be the resulting graph; note that the nodes in S are not contracted. Compute the maximum flow from u to v in $G(S)$. Let $f(u, v)$ be the value of the (u, v) flow, and let $C(u, v)$ be a minumum cut between u and v in $G(S)$. Let S_u be the supernode containing the nodes of G in S which fall on the u side of $C(u, v)$, and let S_v be the supernode containing the remaining nodes of S. Modify T by replacing supernode S with S_u and S_v, connected by an edge of weight $f(u, v)$. Any edge (S, S') incident with S in T is now moved to be incident with S_u if S' is in a contracted node of $G(S)$ on the u side of $C(u, v)$, and is moved to be incident with S_v if S' is in a contracted node of $G(S)$ on the v side of $C(u, v)$; note that the weights of all the edges remain unchanged, including those edges which were moved.

The existence of noncrossing cuts, stated earlier in Lemma 1, provides justification for the contraction operation in the GH method. That is, in order to find a minimum (u, v) cut in G, it is permissible to contract Y; a minimum (u, v) cut in the graph with Y contracted defines a minimum (u, v) cut in G, and of course, the two cuts have the same capacity. Applied iteratively from the leaves of T to S, the lemma can be used to show that a minimum (u, v) cut (for u and v in S) in the contracted graph $G(S)$, has the same *capacity* as a minimum (u, v) cut in G. Such a cut will of course not cross any previously found cuts, and is desired in the GH method because it is then easy to see how to use that cut to split S and how to reconnect the supernode neighbors of S to S_u and S_v.

3.2. Crossing cuts can be used to split a supernode. Consider the basic step in the GH method of dividing a supernode S by computing a minimum cut $C(u, v)$ between

u and v in the contracted graph $G(S)$. This step does two things: it decides how to split S into two new supernodes S_u and S_v, and it decides how to reconnect the neighbors of S to the supernodes S_u and S_v. In this section we will show how the GH method can use crossing cuts in carrying out the first decision.

DEFINITION. A pair of nodes (x, y) is called a *cut pair* for an edge e of an intermediate cut tree T if the nodes of G in the two connected components of $T-e$ form a minimum (x, y) cut in G.

For the following lemma, let T be an intermediate tree produced by the GH algorithm, with e an edge in T between two supernodes S and S'. Let (x, y) be a cut pair for edge e, with $x \in S$ and $y \in S'$; let u and v be any nodes in S, and let $C(u, v)$ be a minimum (u, v) cut in the contracted graph $G(S)$ defined from $T-S$. Let S_u and S_v be the new supernodes created from S, and let $\bar{T}$ be the updated intermediate tree given by the GH algorithm.

LEMMA 4 [GH].[2] *The pair (u, v) is a cut pair for the edge between S_u and S_v, in $\bar{T}$. Assume $x \in U$ (the case when $x \in V$ is symmetric). If (S', S_u) is an edge in $\bar{T}$, then (x, y) is a cut pair for it, and if (S', S_v) is an edge in $\bar{T}$, then (v, y) is a cut pair for it, in $\bar{T}$.*

Initially we will need only the following simpler version of Lemma 4, which follows easily by induction on the number of iterations of the GH algorithm.

COROLLARY 3 [GH]. *Let T be an intermediate tree in the computation of a* GH *cut tree, and let e be an edge in T between two supernodes S and S'. Then there is a pair of nodes (x, y) with $x \in S$ and $y \in S'$ such that (x, y) is a cut pair for e.*

Lemma 4 and its corollary are not as simple as they might at first seem, since x and y may not be the nodes used in the flow that created e, and the nodes that were used might not be in the current supernodes S or S' in $\bar{T}$.

We are now ready for the major theorem of this section.

THEOREM 2. *Let u and v be two nodes of G in supernode S of an intermediate* GH *tree T. If (U, V) is any minimum (u, v) cut in G (with $u \in U$ and $v \in V$), then there exists a minimum (u, v) cut (C_u, C_v) in the contracted graph $G(S)$ (with $u \in C_u$ and $v \in C_v$) such that $S \cap U = S \cap C_u$ and $S \cap V = S \cap C_v$, and such that the capacities of the two cuts are the same.*

Hence to determine how S could be split in a step of the GH method, we need not compute a cut in the contracted graph $G(S)$, but rather use the split of S created by a minimum cut splitting S in the original graph G.

Proof of Theorem 2. By Corollary 3, for each i from 1 to k, $C_i = (G - N_i, N_i)$ is a minimum cut separating some node in $G - N_i$ from some node in N_i, since $S \subseteq (G - N_i)$.

We now apply Corollary 1 to cuts C_1 and (U, V). Corollary 1 implies that there is a minimum (u, v) cut (U_1, V_1) with the same capacity as (U, V), such that $N_1 \subseteq U_1$ or $N_1 \subseteq V_1$, and such that $(G - N_1) \cap U = (G - N_1) \cap U_1$. Since $S \subseteq (G - N_1)$, it follows that $S \cap U = S \cap U_1$ (and $S \cap V = S \cap V_1$).

Now consider the cut $C_2 = (G - N_2, N_2)$. Since N_1 and N_2 are disjoint, and $S \subseteq G - N_2$, it follows that $S \cup N_1 \subseteq G - N_2$. Hence, by Corollary 1 there is a minimum (u, v) cut (U_2, V_2) derived from cuts C_2 and (U_1, V_1) such that

1. (U_2, V_2) has the same capacity as (U_1, V_1) and hence as (U, V).
2. $N_2 \subseteq U_2$ or $N_2 \subseteq V_2$.

[2] As with Lemma 1, the statement and proof of Lemma 4 is found in the body of a proof of a different proposition in [GH], [FF], and [H2]. The simplest such proof of Lemma 4 appears in [FF, p. 182] or [H2, pp. 71–73].

3. $(G - N_2) \cap U_2 = (G - N_2) \cap U_1$, so $N_1 \subseteq U_2$ or $N_1 \subseteq V_2$ and $N_1 \subseteq U_2$ if and only if $N_1 \subseteq U_1$.
4. $S \cap U_2 = S \cap U_1 = S \cap U$ (and $S \cap V_2 = S \cap V$).

Continuing in this way, using the fact that N_i is disjoint from S and from each $N_j : j \leqq i-1$, we can inductively apply Lemma 1 to cuts C_i and (U_{i-1}, V_{i-1}) (the cut obtained in iteration $i-1$) to obtain a minimum (u, v) cut (U_i, V_i) with the properties that

1. (U_i, V_i) has the same capacity as (U, V).
2. $S \cap U_i = S \cap U$ (and $S \cap V_i = S \cap V$).
3. $(G - N_i) \cap U_i = (G - N_i) \cap U_{i-1}$, so for all $j \leqq i$, $N_j \subseteq U_i$ or $N_j \subseteq V_i$ and $N_j \subseteq U_i$ if and only if $N_j \subseteq U_j$.

We conclude then that $S \cap U_k = S \cap U$ (and $S \cap V_k = S \cap V$), and that for each $i \leqq k$, $N_i \subseteq U_k$ or $N_i \subseteq V_k$, and (U_k, V_k) has the same capacity as (U, V). Now since each N_i is strictly on one side or the other of (U_k, V_k), it clearly defines a (u, v) cut (C_u, C_v) in $G(S)$ of the same capacity, and the theorem is proved. □

COROLLARY 4. *For all j, $N_j \subseteq U_k$ if and only if $N_j \subseteq U_j$.*

This corollary, and the last part of line labeled 3 above are not needed in the proof of Theorem 2, but will be needed later.

3.3. Reconnection despite crossing cuts. Theorem 2 shows how to determine, using the original G instead of a contracted graph, a split of S that the GH algorithm could have found. However, a minimum (u, v) cut C in G might split a set N_i between the u and v sides of C (i.e., might cross a previous cut); the GH algorithm has no rules to deal with such cuts. In this section we will see how to use crossing cuts to reconnect the neighbors of S to S_u and S_v.

3.3.1. Modifying the GH cut tree method. We first modify the GH method so that in every intermediate tree, every supernode S contains exactly one node called the *representative* of S, denoted $r(S)$. We start by arbitrarily declaring some node to be the representative of the first supernode of the GH method (the set of all nodes of G). We then impose the rule that when any supernode S is to be split, the flow computed must be between $r(S)$ and some other node v of S. After S is split into two supernodes $S_{r(S)}$ and S_v, $r(S)$ is the representative of $S_{r(S)}$, and v becomes the representative of S_v. It is then easy to see inductively that each supernode has exactly one representative. With this modification, successive application of Lemma 4 yields Lemma 5.

LEMMA 5. *Let T be an intermediate cut tree with S and S' any two adjacent supernodes in T; let N_i be the connected component of $T-S$ containing S'. Then $(G - N_i, N_i)$ is a minimum cut in G separating $r(S)$ and $r(S')$. That is, $(r(S), r(S'))$ is a cut pair for the edge in T between S and S'.*

For the statement of the following theorem, let S and N_j for $j \leqq k$ be as in Theorem 2, and for $j \leqq k$, let $y_j \in N_j$, $x_j \in (G - N_j)$ be such that $(G - N_j, N_j)$ is a minimum (x_j, y_j) cut in G (by Corollary 3, such an (x_j, y_j) exists). Also, for u and v in S, let (U, V) be *any* minimum (u, v) cut in G, and let (U_k, V_k) be the minimum (u, v) cut obtained from (U, V) as in the proof of Theorem 2.

THEOREM 3. *For a fixed j, if $x_j = u$, then $N_j \subseteq U_k$ if and only if $y_j \in U$.*

Proof. Corollary 4 says that $N_j \subseteq U_k$ if and only if $N_j \subseteq U_j$. So all that must be proved is that $N_j \subseteq U_j$ if and only if $y_j \in U$, assuming that $u = x_j$. Now if $u = x_j$, then $x_j \in U_{j-1}$ (since $S \cap U = S \cap U_{j-1}$), so Lemma 1 says that $y_j \in U_j$ if and only if $y_j \in U_{j-1}$. But $y_j \in N_j \subseteq (G - N_{j-1})$, and $(G - N_{j-1}) \cap U_{j-1} = (G - N_{j-1}) \cap U_{j-2}$, so $y_j \in U_{j-1}$ if and only if $y_j \in U_{j-2}$. Now $y_j \in (G - N_p)$ for all $p < j$, so we can induct as above to get

$(G-N_p)\cap U_p=(G-N_p)\cap U_{p-1}$, so $y_j\in U_p$ if and only if $y_j\in U_{p-1}$ for all $p<j$. Hence, assuming that $x_j=u$, it follows that $y_j\in U_k$ if and only if $y_j\in U$, and otherwise, $y_j\in V_k$. □

Theorem 3 is the key to reconnecting neighbors of S after S is split by a crossing cut.

COROLLARY 5. *For S a supernode in an intermediate tree T produced by the modified* GH *method, and for $v\neq r(S)$, let (U, V) be any minimum $(r(S), v)$ cut in G. The following rule correctly decides whether a neighbor of S, S', in T should be connected to $S_{r(S)}$ or to S_v: If $r(S')$ is on the $r(S)$ side of (U, V), then connect S' to $S_{r(S)}$, else to S_v.*

Proof. By Lemma 5, when the modified GH method is used, $r(S)$ satisfies the conditions required of x_j, namely, that $r(S)\in G-N_j$ the cut $(N_j, G-N_j)$ is a minimum $(r(S), r(S_j))$ cut, where S_j is the supernode neighbor of S in N_j. Furthermore, in the modified GH method, $u=x_j=r(S)$ for every j. Hence Theorem 3 implies that there exists a minimum (u, v) cut (U_k, V_k) in $G(S)$ such that for every j, $N_j\subseteq U_k$ if and only if $r(S_j)\in U$. Such a cut (U_k, V_k) could have been computed by the GH algorithm, and so the corollary follows. □

3.3.2. The method in brief. Theorem 2 and Corollary 5 form the basis of our simple version of the GH method. Initially, node 1 is the representative of the supernode consisting of all the nodes. When splitting a supernode S, compute an *arbitrary* minimum cut in G between $r(S)$ and any other node v in S. The nodes of S which fall on the v side of the cut form a new supernode S_v with representative v, and the other nodes in S remain in $S_{r(S)}$ with representative $r(S)$; if S' is a supernode neighbor of S in T before the split, and $r(S')$ falls on the v side of the cut, then replace the (S, S') edge with edge (S_v, S').

3.4. A simple complete cut tree program. To demonstrate the simplicity of our version of the GH method, we give the following program to compute a GH cut tree of input graph G. Theorem 2 and Corollary 5 allow great flexibility in the order in which supernodes are split, but for simplicity, the program below chooses s nodes in order from 2 to n. As in program EQ, p is an n length vector initialized to 1. At iteration s, $p[s]$ is the representative of the supernode that s is in. The edges of T are the final pairs $(i, p[i])$ for i from 2 to n, and edge $(i, p[i])$ has value $fl(i)$. If each edge is considered a directed edge from i to $p[i]$, then T forms a directed tree where every node leads to node 1.

CUT TREE PROGRAM MGH.

```
for s:=2 to n do
 begin
  Compute a minimum cut between nodes
  s and t:=p[s] in G; let X be the set of nodes on the s side
  of the cut. Output the maximum s, t flow value f(s, t).
 fl[s]:=f(s, t);
 for i:=1 to n do
  if (i<>s and i is in X and p[i]=t) then p[i]:=s;
 if (p[t] is in X) then
  begin
  p[s]:=p[t];
  p[t]:=s;
  fl[s]:=fl[t];
  fl[t]:=f(s, t);
  end;
 end;
```


We use the convention that the name of a supernode is given by the name of its representative, and note that after iteration $i-1$, nodes 1 through i are representatives of supernodes, and no node $j>i$ is a representative node in supernode $p[j]$; so for every node $j>s$, $p[v]$ indicates the representative of the supernode that v is in. Every supernode other than 1 points (with the p vector) to exactly one other supernode, and hence if x is a supernode other than 1, then its neighbors consist of those supernodes pointing to x, plus $p[x]$, the supernode to which x points. The neighbors of supernode 1 are just those supernodes with p value 1, i.e., those supernodes that point to 1. During the ith iteration, node $i+1$ becomes the representative of a supernode labeled $i+1$, and all representatives which point to $p[i+1]$ and which fall on the $i+1$ side of the $(i+1, p[i+1])$ cut are now made to point to $i+1$. Since the intermediate trees are being kept in an n-length vector, not an adjacency list, the only subtle part of the program occurs after a flow from $s=i+1$ to $t=p[i+1]$ if t points to a supernode neighbor x of t, and x falls on the s side of the (s, t) cut. In that case we make t point to s, and s point to x; otherwise, s remains pointing to t.

To explicitly accumulate the maximum flow values between all the pairs, we simply add the same two lines of code shown after algorithm EQ; the lines are added just before the final *end.* This is correct, because the set of (s, t) flow pairs generated in MGH is clearly a set that could have been generated in EQ. This accumulation of flow values can also be shown to be correct strictly in the context of the GH method, but was not obvious and was observed only after the discovery of algorithm EQ. Without this observation, a simple $O(n^2)$ method to explicitly calculate the $n(n-1)/2$ flow values is to do depth first search on the final cut tree, so that when backing up from a node x to y, the flow $fl(y, z)$ from y to a descendent z of x can also be computed as the minimum of $fl(x, y)$ and $fl(x, z)$. While this depth first search is not difficult, it requires a change in how T is represented, and the above two-line approach is certainly much simpler.

Note that, as in Algorithm EQ, the only interaction with G is in the minimum cut routine, so the tree could be inferred from $n-1$ calls to an oracle which returns a minimum cut and its value.

Relation with Algorithm EQ. The modified GH method can be described in terms of Algorithm EQ. To compute the GH tree, change step 4 of Algorithm EQ to read:

4. For every node i *other than* s, if i is a neighbor of t, and i is on the s side of (X, Y), then modify T' by disconnecting i from t, and connecting i to s, *labeling the new* (i, s) *edge with the label from the old* (i, t) *edge.*

Phrases in italics show the differences between this step 4 and the step 4 of Algorithm EQ.

4. Additional comments and extensions. (1) It is easy to underestimate the amount of programming detail needed by the original GH method. In fact, the ideas leading to this paper partly began after a failed attempt to quickly implement the method. The implementation was made more difficult because we used existing code for finding the maximum flow, but we did not understand the code well, and we needed to modify it to implement graph contraction and expansion. With the modified GH method of this paper, we totally avoid these difficulties, since we never touch any of the existing code, and never touch the graph after it is input.

In addition to the obvious work involved in contraction, an implementation of the original GH method must do a fair amount of work implied by the need to do

contraction. It must maintain T in a way so that the connected components can be efficiently found, and so that the nodes of G contained in particular supernodes of T can be identified, both to split a supernode, and to properly contract the nodes of G contained in a component of $T-S$. It must also maintain information about the connected components of $T-S$, or it must reexpand components after a flow, so that it can determine which supernodes fall on the u side and which on the v side of the cut $C(u, v)$ in $G(S)$.

(2) The original GH method might run faster in practice than the modified method (although the worst case asymptotic time is the same), since the contracted graphs are smaller than the original graph. However, it is an empirical question whether the speedup in flow computation compensates for the work needed to implement contraction and all the associated work implied by contraction; contraction should be seen as a heuristic that might accelerate the performance of the program.

(3) Some of the ideas in this paper have been extended and used to study the structure of minimum cuts in three other settings. A GH cut tree represents at least *one* minimum cut for each pair of nodes in an undirected edge-weighted graph. In [GN1] we generalize the GH cut tree, showing how to efficiently and compactly represent *all* minimum cuts between each pair of nodes. Interestingly, our method is based on equivalent flow trees, rather than on cut trees, further extending the importance of efficient computation of equivalent flow trees. This work also connects to and builds on recent work by Matula [M] and by Mansour and Schieber [MS] on computing connectivity quickly. In related work [GN2] we show how to construct with $O(n)$ maximum flow computations a cut tree for weighted node cuts, rather than edge cuts. We also show how to compactly represent weighted edge cuts in a directed graph.

(4) Very recently, Cheng and Hu [CH] have further reduced the importance of noncrossing cuts in equivalent flow trees. In Algorithm EQ and in the algorithm from [GrH], crossing cuts are allowed, but the proofs of correctness still use the fact that noncrossing cuts exist. Cheng and Hu give a different method which uses only $n-1$ maximum flow computations, and can be used to produce equivalent flow trees, but not cut trees. However, its proof of correctness does not even depend on the existence of noncrossing cuts. Because of that, their method can be used to represent minimum cut values where the value of a cut is given by an arbitrary function, i.e., is not the sum of the edge capacities crossing the cut. It is not difficult then to use this method to improve the problem considered in Schnorr [SC]. For a pair of nodes (i, j) define $\beta(i, j)$ as the minimum of the flow in a directed graph from i to j, or from j to i. These β values are needed in several problems [GN2], [GU]. Schnorr shows, using a very clever idea, that all the pairwise β values can be computed with $O(n \log n)$ maximum flow computations on the original graph. He then modifies that method to show that, with contraction, those $O(n \log n)$ flows run in total time $O(n^4)$. However, using the method of [CH] with its relaxed notion of cut values, the β values can be computed using only $O(n)$ maximum flow computations [GN2]. Hence in Schnorr's problem, contraction can also be avoided without sacrificing efficiency.

5. Conclusion. We have shown how to efficiently construct equivalent flow trees and GH cut trees without finding or maintaining noncrossing cuts, hence without node contraction and its associated work. The main theoretical consequence is conceptual clarity: node contraction, which is presented in existing discussions of the GH method as the fundamental algorithmic idea, is in fact not fundamental to cut tree computation; it should be seen as a heuristic which might accelerate the running of the flow computations. Similarly, although the *existence* of noncrossing cuts remains central in the *logic* of cut trees, they are not explicitly needed in the efficient *computation* of cut trees. An additional theoretical consequence is the fact that a cut tree can be inferred from $n-1$ queries of an oracle which alone knows the actual graph. On the practical side, the import of these observations is that they lead to very simple, efficient programs for computing equivalent flow trees and cut trees; most of the programming and data structure details of the original GH method become unnecessary when contraction is avoided.

REFERENCES

[AH] D. ADOLPHSON AND T. C. HU, *Optimal linear ordering*, SIAM J. Appl. Math., 25 (1973), pp. 403–423.

[AMS] S. AGARAWAL, A. K. MITTAL, AND P. SHARMA, *Constrained optimum communications trees and sensitivity analysis*, SIAM J. Comput., 13 (1984), pp. 315–328.

[CH] C. K. CHENG AND T. C. HU, *Maximum concurrent flow and minimum ratio cut*, Tech. Report CS88-141, University of California, San Diego, CA, December 1988.

[E] S. E. ELMAGHRABY, *Sensitivity analysis of multi-terminal network flows*, J. ORSA, 12 (1964), pp. 680–688.

[FF] L. R. FORD AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[FR, FR] H. FRANK AND I. T. FRISCH, *Communication, Transmission and Transportation Networks*, Addison-Wesley, Reading, MA, 1972.

[GH] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, SIAM J. Appl. Math., 9 (1961), pp. 551–570.

[GrH] F. GRANOT AND R. HASSIN, *Multi-terminal maximum flows in node capacitated networks*, Discrete Appl. Math., 13 (1986), pp. 157–163.

[GU] D. GUSFIELD, *A graph theoretic approach to statistical data security*, SIAM J. Comput. 17 (1988), pp. 552–571.

[GU1] ———, *Very simple algorithms and programs for all pairs network flow analysis*, Tech. Report cse-87-1, Division of Computer Science, University of California, Davis, CA, April 1987.

[GN1] D. GUSFIELD AND D. NAOR, *Extracting maximal information about sets of minimum cuts*, Tech. Report cse-88-14, Division of Computer Science, University of California, Davis, CA, September 1988.

[GN2] ———, *Generalized cut trees: Efficient algorithms and uses*, Tech. Report cse-89-5, Division of Computer Science, University of California, Davis, CA, March 1989.

[HA] W. HANSJOACHIN, *Ten Applications of Graph Theory*, D. Reidel, Boston, MA, 1984.

[H1] T. C. HU, *Integer Programming and Network Flows*, Addison-Wesley, Reading, MA, 1969.

[H2] ———, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.

[H3] ———, *Optimum communication spanning trees*, SIAM J. Comput., 3 (1974), pp. 188–195.

[HR] T. C. HU AND F. RUSKEY, *Circular cuts in a network*, Math. Oper. Res., 5 (1980), pp. 362–373.

[HS] T. C. HU AND M. T. SHING, *Multiterminal flows in outplanar networks*, J. Algorithms (1983), pp. 241–261.

[L] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[LP] L. LOVASZ AND M. D. PLUMMER, *Matching theory*, Ann. Discrete Math., 29 (1986), North-Holland, Amsterdam, the Netherlands.

[M] D. MATULA, *Determining edge connectivity in $O(nm)$*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, October, 1987.

[MS] Y. MANSOUR AND B. SCHIEBER, *Finding the edge connectivity of directed graphs*, J. Algorithms, 10 (1989), pp. 76–85.

[PG] D. PHILLIPS AND A. GARCIA-DIAZ, *Fundamentals of Network Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[SC] C. P. SCHNORR, *Bottlenecks and edge connectivity in unsymmetrical networks*, SIAM J. Comput., 8 (1979), pp. 265–274.

[S] Y. SHILOACH, *A multi-terminal minimum cut algorithm for planar graphs*, SIAM J. Comput., 9 (1980), pp. 219–224.

[T] L. E. TROTTER, JR. *On the generality of multi-terminal flow theory*, Ann. Discrete Math., 1 (1977), pp. 517–525.

[VL] J. VAN LEEUWEN, *Graph algorithms*, Tech. Report RUU-CS-86-17, Department of Computer Science, University of Utrecht, Utrecht, the Netherlands, October 1986.