BICONNECTED GRAPH ASSEMBLY and RECOGNITION OF DFS TREES

by

Torben Hagerup

A 85/03

February 1985

FB 10, Informatik
Universität des Saarlandes
6600 Saarbrücken
West Germany

Abstract:

We present two new algorithms on undirected graphs. The first of these takes as input a biconnected graph G and produces a list of simple instructions that may be used to build G from a trivial initial graph in such a way that all intermediate graphs are biconnected. Each instruction specifies either 1) the addition of an edge between two nodes, or 2) the addition of a new node "on" an existing edge. We shall say that the algorithm solves the problem of "biconnected graph assembly".

The second algorithm takes as input a connected graph G and a spanning tree T of G given by a marking of the tree edges (i.e., the tree is not rooted). It decides whether there is a depth-first search of G such that the undirected tree implied by the search is identical to T. This may be called "recognition of DFS trees". In fact, the algorithm computes the set of those nodes that may be taken as roots of such a search.

Both algorithms are based on depth-first search and work in linear time and space.

1. BASIC DEFINITIONS AND FACTS

We shall assume familiarity with the most basic definitions concerning undirected graphs and with their common representations in a computer. This material may be found in e.g. (Harary, 1969; Aho et al., 1974; Even, 1979; Tarjan, 1972).

An (unrooted) tree is a graph that is connected and acyclic. In a tree, there is a unique simple path between any two given nodes. A rooted tree is a tree in which one node has been designated as the root. In a rooted tree, one may talk about father and son relationships. The father of a node u is the first node ≠ u on the simple path from u to the root. The root has no father. If u is the father of v, v is a son of u. The reflexive and transitive closure of the father relation gives the ancestor relation and the inverse descendant relation which is the reflexive and transitive closure of the son relation. A leaf is a node which has no sons.

A spanning tree of a graph G is a subgraph of G which is a tree and which includes all the nodes of G. Given a rooted spanning tree T of a graph G, we may classify the edges of G relative to T into 1) tree edges, 2) back edges, i.e. non-tree edges that connect a node u to an ancestor or a descendant of u, and 3) cross edges, the remaining edges. We shall often draw tree edges solid, back edges dashed, and cross edges dotted. When a back edge is being considered from one of its endpoints, it may be characterized as "upward", i.e. going to an ancestor, or as "downward".

By G-{e}, where G is a graph and e an edge of G, we mean the graph obtained from G by deleting e, and G-{u}, where u is a node of G, is the graph obtained from G by deleting u and all edges incident on u. If G is a connected graph, a node u is called an articulation point of G if G-{u} is not connected. A graph is biconnected if it is connected and has no articulation points.

To search a graph is to "visit" all the nodes of the graph, whereby the details of visiting a node may vary. Depth-first search (DFS) is one particular way of searching a graph. For a connected graph, it takes the following form: First, choose and "discover" some initial node. Then, as long as there remains discovered nodes with incident edges that were not yet explored in the direction away from the node in question, choose the most recently discovered such node and explore one of its unexplored edges. "Exploring" an edge incident on some node means to discover the node at the other end of the edge, if it has not already been discovered. Of the computations which together comprise the visiting of a node u, some may be done when u is discovered, and others later when the search returns to u to explore one of its incident edges, or to discover that there are no more unexplored edges. It is the "most recently discovered" above that makes the search "depth-first", and it also shows that the search may be succinctly described as a recursive algorithm:

```
procedure DFS(u:Node);
begin
  (* some computations *)
  for all nodes v adjacent to u
  do if v has not yet been discovered
     then DFS(v);
  (* some computations *)
end;
```

Fig. 1. Generic depth-first search.

Hence, in the terminology of the implementation of recursive procedures, the node currently being visited is the one whose stack frame is currently active (on top of the run-time stack).

One easily proves that each node is discovered exactly once and that each edge is explored exactly once in each direction. If visiting a node takes time proportional to its degree, the whole search takes time $\mathcal{O}(|E|)$, where $|E|$ is the number of edges in the graph. The DFS number of a node u is defined to be k if u is the k'th node to be discovered during the search. DFS numbers of all nodes may easily be computed along with the search.

We may associate a subgraph T of a connected graph G with a DFS of G. T contains all the nodes of G and precisely those edges that lead to new (i.e., not yet discovered) nodes during the search. T is a spanning tree of G. We call T a DFS tree of G and sometimes root T at the initial node of the search. A central observation (Tarjan, 1972; Aho et al., 1974) is that with respect to T, rooted as just described, G contains no cross edges.

2. BICONNECTED GRAPH ASSEMBLY

Problem definition.

Consider the following types of operations on undirected graphs (see fig. 2 below):

E) Join two non-adjacent nodes by an edge.

N) Delete the edge between two adjacent nodes v and w, create a new node and join it to v and w (we may describe this informally as placing a new node "on" the edge (v,w)).

Fig. 2. E and N operations.

If an operation of type E or N is applied to a biconnected graph, it is easy to see that the resulting graph is biconnected. Hence so is any graph that can be obtained from the (biconnected) complete graph K_3 on 3 nodes by applying a sequence of the above operations. Whitney proved that the converse is true as well: Any biconnected graph with at least 3 nodes may be obtained in this way (Whitney, 1932). See fig. 3 below for an example.

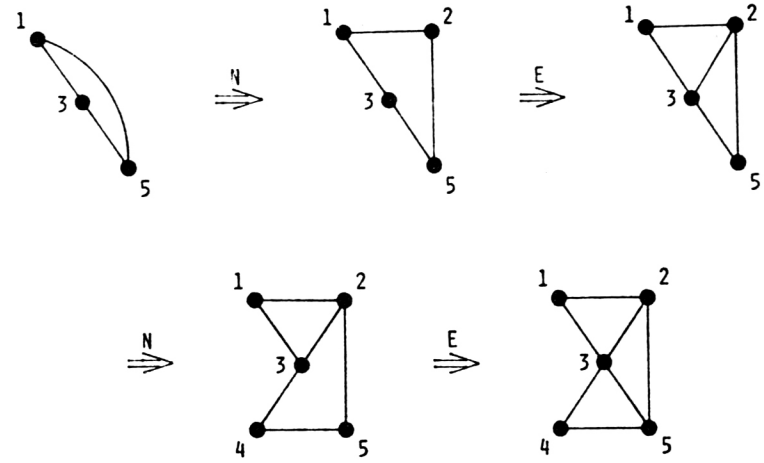

Fig. 3. A biconnected graph assembly.

Whereas Whitney's proof is non-constructive, our algorithm gives an actual derivation of a given biconnected graph G in the form of a list of instructions, each specifying an operation of type E or N, such that if the operations are carried out in the given order, starting from K_3, the graph G results.

Development of the algorithm.

The overall strategy is to generate the instructions in reverse order, namely by taking the given graph G apart piece by piece using the inverses of E and N operations:

E^{-1}) Remove an edge.

N^{-1}) For a node u of degree 2 connected to non-adjacent nodes v and w, replace u and its incident edges by the edge (v,w).

Fig. 4. E^{-1} and N^{-1} operations.

If each inverse operation is chosen so as to preserve biconnectivity, the process must eventually yield K_3 since no other graph of 3 nodes is biconnected (this is very similar to step 3 of the algorithm given in (Valdes et al., 1979) for the recognition of series parallel digraphs). Then, provided that we have saved enough information about the single inverse operations, we may run the whole process in reverse and thereby (re-)construct G in the desired way. Since the latter step is trivial using a stack, it will not be further discussed here, except in an example. The problem which remains is to preserve biconnectivity while E^{-1} and N^{-1} operations are employed to disassemble G.

We establish the validity of the algorithm without reference to Whitney's result, thus obtaining an independent proof of the latter. We first prove a number of lemmas, where each lemma states that certain changes to a biconnected graph do not destroy biconnectivity.

Lemma 1: Consider a DFS $\mathcal{Y}$ of a biconnected graph G with at least 3 nodes. Let u be a leaf in the rooted tree associated with $\mathcal{Y}$, and let a be the node adjacent to u of minimal DFS number. Then all back edges (u,v) except (u,a) may be removed without destroying biconnectivity (see fig. 5).

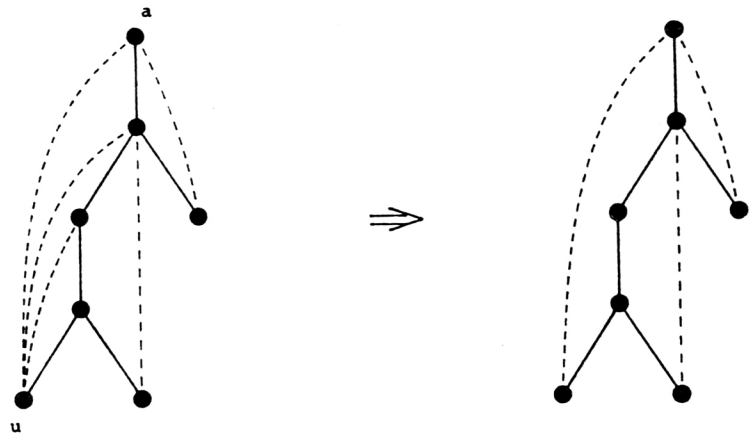

Fig. 5. The transformation of lemma 1.

Proof: Observe first that a exists and that (u,a) is a back edge since G is biconnected. Let H be the graph resulting from the transformation of the lemma. It is easy to see that there is a DFS of H that produces the same tree and assigns the same DFS numbers as $\mathcal{Y}$. Hence we may refer to these without ambiguity. We now use a well-known characterization of articulation points in terms of DFS trees which may be found in e.g. (Tarjan, 1972; Aho et al., 1974), where it is used in an algorithm for finding the biconnected components of a graph. Define on the set of nodes a function LOW given by

LOW(v) = min({DFSnumber(v)} ∪
{DFSnumber(w) | w is connected to a descendant of v})

Then a node v is an articulation point exactly if one of the following holds:

1) v is the root and has more than one son,

or

2) v is not the root and has a son s with
LOW(s) ≥ DFSnumber(v) .

The final step of the proof is to note that LOW(v) computed in H is exactly the same as LOW(v) computed in G, for all nodes v. Hence, by the above characterization, H has no articulation points.

Lemma 2: Let G be a biconnected graph of at least 4 nodes, and u, v and w three nodes of G that form a clique (i.e., there is an edge between any two of them). If u is of degree 2, the edge (v,w) may be removed without destroying biconnectivity (see fig. 6).

Fig. 6. The transformation of lemma 2.

Proof: Let H be the graph G-{(v,w)}. It suffices to show the existence of a cycle in H containing v and w. But there must be a node z∈{u,w} connected to v, since otherwise w would be an articulation point of G. On the other hand, v is not an articulation point either, so there must be a simple path from z to w which avoids v (and hence u) (see fig. 7). This path completes the desired cycle.

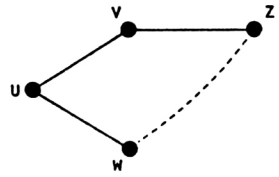

Fig. 7. Illustration of the proof of lemma 2.

Lemma 3: Let G be a biconnected graph, and u, v and w three nodes of G such that G contains the edges (u,v) and (u,w), but not (v,w), and such that u has degree 2. Then the node u and its incident edges (u,v) and (u,w) may be replaced by the edge (v,w) without destroying biconnectivity (see fig. 8).

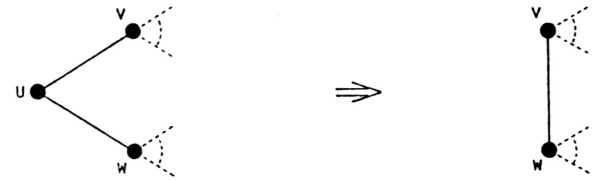

Fig. 8. The transformation of lemma 3.

Proof: There is a path in G from v to w that avoids u. But then clearly the graph resulting from the change contains a cycle passing through v and w.

We are now ready to describe a subroutine DeleteLeaf(u) which is the work-horse of our algorithm. It assumes that we have found a leaf u in the DFS tree of a graph which is either the given graph G or an intermediate graph. Let f be the father of u, and a the neighbouring node of minimal DFS number. DeleteLeaf(u) now consists of three steps:

1) Remove all back edges (u,v) except (u,a).

These are E^{-1} operations, and lemma 1 assures us that biconnectivity is preserved. Now u is of degree 2.

2) If the edge (f,a) is present, remove it.

Again, an E^{-1} operation. Biconnectivity of the resulting graph is guaranteed by lemma 2.

Whether or not any action was taken in step 2, lemma 3 now applies. Hence we may

3) Replace the node u and its incident edges by the edge (f,a).

This is an N^{-1} operation.

The net result of the call DeleteLeaf(u) is that some back edges have been removed or inserted, and that u has been deleted. Note that what is left of the original DFS tree is a DFS tree of what remains of the graph. Hence we may continue to call DeleteLeaf at another leaf, etc. This may be elegantly formulated as a depth-first search:

```
procedure Eliminate(u:Node);
begin
  Recursively eliminate all sons of u;
  (* u now is a leaf *)
  DeleteLeaf(u);
end;
```

Fig. 9. High-level description of the algorithm.

The validity of the comment, which is crucial to correctness, is easily proved by induction.

An example.

Let us apply the algorithm to the example graph below (fig. 10).

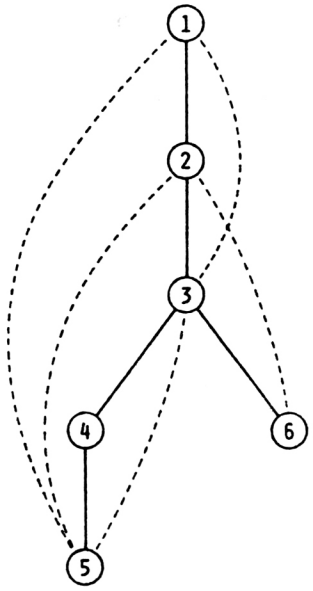

Fig. 10. Example biconnected graph.

Assume that sons are visited in the order from left to right. Then the first leaf to be discovered is node 5. f and a are in this case the nodes 4 and 1, respectively. There are three back edges incident on 5, so we remove two of them, (5,2) and (5,3). Nodes 4 and 1 are not connected, so nothing happens in step 2 of DeleteLeaf(5). Finally, node 5 and its incident edges (5,4) and (5,1) are replaced by the edge (1,4).

The search now returns to node 4, which has become a leaf. There is only one back edge incident on 4, so nothing is done in step 1. Since 3 and 1 are connected, we remove the edge (1,3) in step 2. In step 3, node 4 and its incident edges are removed, and (1,3) (re-)inserted.

The search next returns to 3 and continues to 6, which is a leaf. (2,3) is removed, and 6 and its incident edges replaced by (2,3). What is left is the complete graph on nodes 1, 2 and 3.

The instructions output by the algorithm during the above execution might be as in the left column below (fig. 11). The right column shows the instructions read backwards and re-interpreted as instructions to construct the graph from K_3.

Remove edge (5,2)	Start with the complete graph on nodes 1, 2 and 3.
Remove edge (5,3)	Add new node 6 on edge (2,3)
Remove node 5 between nodes 1 and 4	Add edge (2,3)
Remove edge (1,3)	Add new node 4 on edge (1,3)
Remove node 4 between nodes 1 and 3	Add edge (1,3)
Remove edge (2,3)	Add new node 5 on edge (1,4)
Remove node 6 between nodes 2 and 3	Add edge (5,3)
This leaves the complete graph on nodes 1, 2 and 3.	Add edge (5,2)

Fig. 11. Output of the algorithm on input as in fig. 10.

The stacks of descendants.

In order to obtain an algorithm which works in linear time and space, one further complication has to be introduced. The test in step 2 of DeleteLeaf must be carried out in constant time. Whereas this is easy if the graph is represented by an adjacency matrix, the latter data structure in general takes up too much space, and we shall assume instead that the graph is given as a set of adjacency lists. But then the obvious algorithm for testing whether an edge is present takes more than constant time. We solve the problem in the following way:

Each node in the graph has an associated stack of nodes called the stack of descendants. While the search is at some node u (the call Eliminate(u) is the last to have been initiated and not yet completed), the stack of some other node a is empty if u is not a descendant of a, and otherwise contains exactly those nodes that 1) lie on the unique simple tree path from a to u, and 2) are connected to a. The nodes occur in the same order as on the path from a to u, with the node closest to u immediately accessible. The stacks may easily be maintained during the search: When the search proceeds to a new node u, u is pushed on the relevant stacks which may be found by following the upward back edges incident on u, and when the search leaves u to return to u's father, u is popped from the same stacks.

The test "Are f and a connected?" in step 2 now simply becomes "Is f at some fixed position (first or second, depending on programming details) near the top of a's stack of descendants?". We are still unable to actually locate and delete the edge (f,a) in constant time but fortunately, this turns out not to be needed since (f,a) is inserted again in step 3 anyway. See the complete algorithm below (fig. 12) for details.

The complete algorithm.

Input: A biconnected graph G.

Output: Instructions describing how G may be disassembled by E^{-1} and N^{-1} operations that preserve biconnectivity. Read in reverse order and re-interpreted as in the example above, the instructions specify a biconnected assembly of G.

Calling conventions: The procedure Disassemble (fig. 12) should be called on two nodes as actual parameters, where the first one will be taken as the root, and the second one is ignored (generally, this parameter is the father of the first parameter). Before the call, New[u] should have the value true for all nodes u, and all stacks of descendants should be empty. Count should be initialized to the value 1, and NodesLeft to |V|, the number of nodes in the graph. The latter variable is used to detect when the graph has been reduced to K_3.

```
procedure Disassemble(u,f:Node); (* f is the father of u *)
begin
  New[u]:=false;
  DFSnumber[u]:=Count;
  Count:=Count+1;
  for all nodes v adjacent to u
  do if not New[v] (* v is an ancestor of u *)
     then push u on v's stack of descendants;
  for all nodes v adjacent to u
  do if New[v]
     then Disassemble(v,u);
  (* u's adjacency list will grow during this process;
     only nodes initially adjacent to u need be iterated over *)
  for all nodes v adjacent to u
  do if DFSnumber[v]<DFSnumber[u] (* v is an ancestor of u *)
     then pop (* u *) from v's stack of descendants; (* undo *)
  if NodesLeft>3 (* more to do *)
  then
    begin (* DeleteLeaf(u) *)
      a:=lowest-numbered node adjacent to u;
      (* beginning of step 1 *)
      for all nodes v adjacent to u
      do if DFSnumber[a]<DFSnumber[v]<DFSnumber[f]
            (* a back edge - but not (u,a) *)
         then
           begin
             Delete the edge (u,v);
             writeln('Remove edge ',(u,v));
           end;
      (* beginning of steps 2 and 3 *)
      if f is on top of a's stack of descendants
         (* f and a are connected *)
      then writeln('Remove edge ',(f,a)) (* but don't do it *)
      else
        begin
          Insert the edge (f,a);
          Push f on a's stack of descendants;
        end;
      Delete the edges (u,f) and (u,a);
      writeln('Remove node ',u,' between nodes ',f,' and ',a);
      (* end of steps 2 and 3 *)
      NodesLeft:=NodesLeft-1;
    end
  else writeln(u,' is one of the 3 nodes left in the K3 graph');
end; (* Disassemble *)
```

Fig. 12. Detailed description of the algorithm.

Complexity analysis.

The space taken up by a node consists of some fixed-sized fields, the adjacency list and the stack of descendants. Although the adjacency list of a node u may grow during the execution of the algorithm due to the edge insertions in step 3 of DeleteLeaf, its size will never double and so is still $\mathcal{O}$(degree(u)), where degree(u) means the original degree of u. u's stack of descendants certainly grows no bigger than this since all nodes in the stack are also adjacent to u (to take advantage of this, the stacks are most conveniently implemented as linked lists). Hence the total space requirements of node u are $\mathcal{O}$(degree(u)). Summing over all nodes gives a space complexity of $\mathcal{O}$(|E|).

By the above, clearly the time spent visiting u is $\mathcal{O}$(degree(u)). Hence the total execution time is also $\mathcal{O}$(|E|).

Implementation considerations.

We first show that one may avoid explicitly storing the stacks of descendants. The stack of node u never contains nodes that are not also in u's adjacency list. Hence the adjacency list could answer the same queries as the stack if it happened to always be ordered conveniently. But one may arrange for this to happen. "Push v on u's stack" then becomes "delete v from wherever it occurs in u's adjacency list, and re-insert it at the front". "Pop a node from u's stack" becomes "move the node at the front of u's adjacency list to the end of the list". By inspecting the algorithm, one may check that wherever a push is required, an available pointer allows it to be performed in constant time (for pop, this is trivial).

Secondly, it has so far been tacitly assumed that the adjacency lists were doubly-linked and contained pointers linking the occurrence of v in u's adjacency list with the occurrence of u in v's adjacency list, for each edge (u,v). In fact, one may make do with singly-linked adjacency lists and no such cross references. The reason is that although the algorithm "claims" to delete various edges, it isn't ever necessary to actually delete an edge. If an edge, supposed to have been deleted, is later again inspected, the workings of the algorithm are such that no action is taken. The linear time and space bounds still hold with this change.

Finally, it is possible to combine the two modifications above. In this situation, the deletion which was part of the implementation of push above is no longer feasible and is just ignored. Hence the re-insertion part of pop should also be ignored. One is in effect using an initial segment of the adjacency list as the stack of descendants, and a node may occur here as well as later in the adjacency list. Again, one may show that the change is harmless, as far as correctness is concerned. Furthermore, the adjacency lists grow to at most twice their length in the original implementation, and the algorithm still runs in linear time and space.

A modified algorithm using multi-graphs.

If it is acceptable to have graphs with multiple edges (more than one edge between some pair of nodes) occur as intermediate graphs, the stacks of descendants are no longer needed and a simpler solution is possible. To accommodate this case, step 2 of DeleteLeaf should be removed, and in step 3 the edge (f,a) should be inserted even if one or more edges (f,a) are already present. In step 1, if there are several edges (u,a), all but one should be removed together with the other back edges. Finally, when there are only 3 nodes left, the algorithm should continue to remove edges until K_3 is reached.

3. RECOGNITION OF DFS TREES

In the examples below (fig. 13) of two spanning trees (heavily drawn lines) of the same graph G, the first tree is associated with a depth-first search of G, e.g. one that discovers the nodes in the order 1-2-3-4, whereas the second one is not. To see this, note that in a rooted DFS tree, there can be no edges between brothers (sons of the same father).

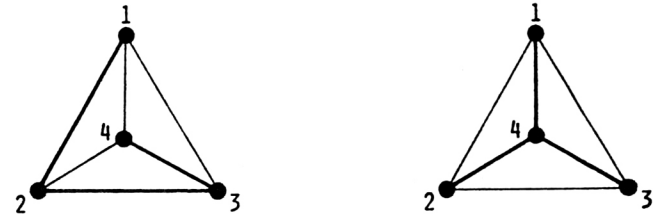

Fig. 13. Two spanning trees of the same graph.

We present an algorithm that takes as input an unrooted spanning tree T of a connected graph G and computes the set of possible DFS roots, i.e. the set of nodes from which it is possible to start a DFS of G whose associated tree is identical to T. In particular, if this set is empty, T is not a DFS tree of G.

The first step towards a solution is the simple observation (Tarjan, 1972, theorem 1) that a spanning tree T is a DFS tree if and only if there are no cross edges with respect to T. We already used a special case of this above. Note carefully, however, that the notion of a cross edge makes sense for a rooted tree only, whereas the given tree is not rooted. Once a root has somehow been decided upon (i.e., a candidate initial node for the DFS has been chosen), one may use the above criterion to test whether there is a DFS starting at that node which is associated with the given tree. We do this by means of a tree search, a search of the graph which is like a DFS except that only tree edges are explored in order to find new nodes (thus the search is forced to follow the given spanning

tree). We say that a tree search discovers the nodes in preorder and accordingly speak of preorder numbers rather than DFS numbers. The number of cross edges may be determined in linear time by a tree search which considers each edge in turn and works directly from the definition: A cross edge is one that does not connect an ancestor and a descendant. During a tree search starting at the chosen root, one may easily arrange to have a field Ancestor associated with each node indicate whether the node is an ancestor of the node currently being visited, in a sense by simulating the run-time stack associated with the recursive procedure calls. Thus upward back edges may be detected, and if the algorithm is made to ignore non-tree edges leading to nodes having a higher preorder number than the node currently being visited, each non-tree edge is considered exactly once and may at that point be classified as a back edge or as a cross edge. The details are given below (fig. 14).

Calling conventions: Fields PreNumber and FatherLink are assumed to indicate for each node its preorder number and the edge connecting it to its father, respectively. For each edge e, a boolean field TreeEdge[e] specifies whether e is part of the given spanning tree. All fields Ancestor must be initialized to the value false, and CrossCount to 0. Then after a call CountCrossEdges(u), where u is the chosen root, CrossCount will contain the number of cross edges.

```
procedure CountCrossEdges(u:Node);
begin
  Ancestor[u]:=true;
  for all edges e=(u,v) incident on u
  do if e≠FatherLink[u] (* v is not the father of u *)
     then
       if TreeEdge[e]
       then CountCrossEdges(v)
       else
         if (PreNumber[v]<PreNumber[u]) and not Ancestor[v]
         then CrossCount:=CrossCount+1;
  Ancestor[u]:=false;
end;
```

Fig. 14. Algorithm to count cross edges.

Definition: For each node u, let T_u be the given spanning tree, but rooted at u, and let C(u) be the number of cross edges with respect to T_u.

The algorithm CountCrossEdges described above computes C(u) for a given node u in time $\mathcal{O}(|E|)$. Hence the set $\{u|C(u)=0\}$, which is the desired final answer, may be computed in time $\mathcal{O}(|V||E|)$ (here |V| is the number of nodes) by repeated application of CountCrossEdges on all possible roots. Being more clever, we can reduce this to $\mathcal{O}(|E|)$.

The first idea is to consider, for each tree edge e=(u,v), the quantity

$$D[e] = \pm(C(v)-C(u))$$

which indicates by how much the value of C changes when the root is moved a small distance in the graph. Since e is undirected, we must find a consistent way of defining the sign of D[e]. Consider for this purpose an arbitrary, but henceforth fixed, root Root and the associated rooted tree T_{Root}. In T_{Root}, u is the father of v, or vice versa.

Definition: For each tree edge e=(u,v), where u is the father of v in T_{Root}, let

$$D[e] = C(v)-C(u) \quad .$$

Each tree edge e=(u,v), where u is the father of v in T_{Root}, may be considered as follows to partition the nodes of G into two groups: The removal of e splits the given spanning tree into two connected components. Let V_1 be the set of nodes connected to u in T-{e}, and let V_2 be the set of remaining nodes. See fig. 15.

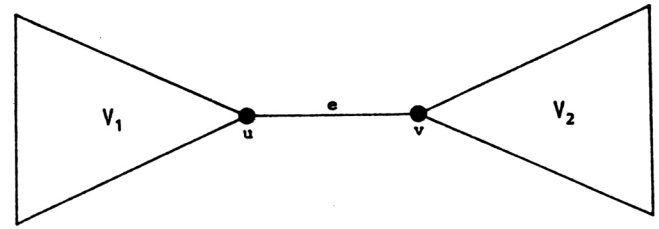

Fig. 15. Node partition relative to a tree edge.

Now consider a (non-tree) edge $e'=(x,y)$, where $x \in V_1 \setminus \{u\}$ and $y \in V_2 \setminus \{v\}$. e' is a cross edge relative to T_u as well as relative to T_v. Hence we may say that e' cancels out and contributes 0 to D[e]. Likewise, an edge (x,y), where either $\{x,y\} \subseteq V_1$ or $\{x,y\} \subseteq V_2$, is a cross edge relative to T_u iff it is a cross edge relative to T_v, and also contributes 0 to D[e].

An edge (u,y), where $y \in V_2 \setminus \{v\}$, is a cross edge relative to T_v, but not relative to T_u. It hence contributes +1 to D[e]. Similarly, an edge (x,v), where $x \in V_1 \setminus \{u\}$, contributes -1 to D[e]. This exhausts the possible cases, and we may thus state that

$$D[e] = \#\{(u,y) \mid y \in V_2 \setminus \{v\}\} - \#\{(x,v) \mid x \in V_1 \setminus \{u\}\} \quad .$$

Given a tree edge $e=(u,v)$, it is not clear how to compute D[e] reasonably fast. Even though membership in V_1 and V_2 via some preprocessing may be decided in constant time, it still seems necessary to step through the entire adjacency lists of u and v, which is too expensive for a linear-time algorithm. The second idea is to compute the values of D for all tree edges simultaneously.

Consider a non-tree edge $e'=(x,y)$ and the unique simple tree path p from x to y. It follows from the preceding discussion that e' contributes ± 1 to the D values of precisely two tree edges, namely the first and the last edge on p (whose length is at least 2) (fig. 16).

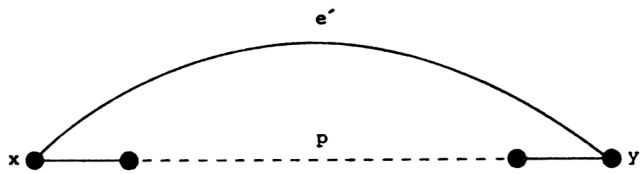

Fig. 16. A non-tree edge and the corresponding tree path.

As for the sign to choose, consider the close-up in fig. 17. By definition, e' contributes +1 to D[e] if u is the father of v in T_{Root}, and -1 otherwise.

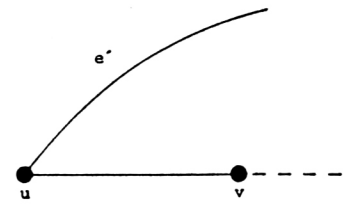

Fig. 17. A non-tree edge and part of the tree path.

The algorithm successively considers all non-tree edges and accumulates their contributions to D values in counters associated with the tree edges. When all edges have been considered, the D value of each tree edge may be read off its counter.

For a non-tree edge (x,y), there are two possibilities (fig. 18 below):

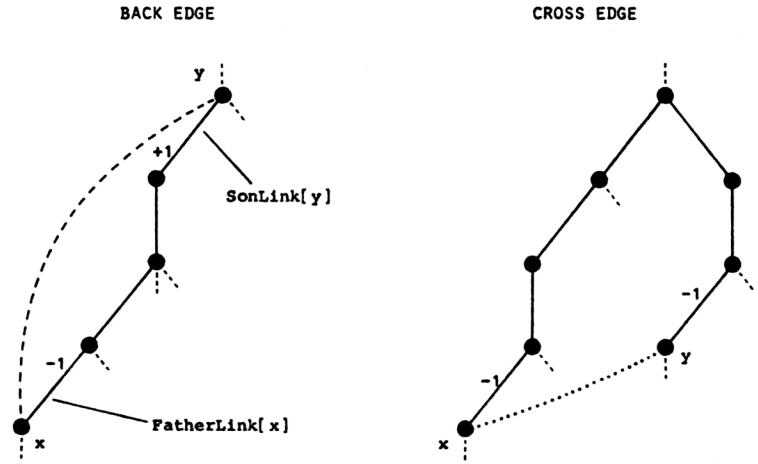

Fig. 18. The two possible cases for a non-tree edge (x,y).

1) (x,y) is a back edge in T_{Root} with, say, y an ancestor of x. Then the tree path from x to y is the obvious one, always

going from son to father. The edge connecting x to its father (FatherLink[x]) receives a contribution of -1, and the edge connecting y to its son in the direction of x (the son which is an ancestor of x) receives a contribution of +1.

2) (x,y) is a cross edge in T_{Root}. Then the tree path between x and y goes upward from each of x and y until it reaches their lowest common ancestor. In any case, FatherLink[x] and FatherLink[y] both receive a contribution of -1.

We have already seen how to distinguish between back edges and cross edges during a tree search. The only problem which still has to be considered is how to find the edge linking y to its son in the direction of x in 1) above. But this is easily solved by associating with each node a dynamically changing field SonLink which gives the edge to the son whose tree of descendants is currently being searched. Details in the algorithm below (fig. 19), which assumes the following

Calling conventions: All Ancestor fields should have the value false, and all D fields the value 0 initially. The outermost call should be Compute_D_values(Root).

```
procedure Compute_D_values(x:Node);
begin
  Ancestor[x]:=true;
  for all edges e=(x,y) incident on x
  do if e≠FatherLink[x] (* y is not the father of x *)
     then
       if TreeEdge[e]
       then
         begin
           SonLink[x]:=e;
           Compute_D_values(y);
         end
       else
         if PreNumber[y]<PreNumber[x]
         then
           begin
             D[FatherLink[x]]:=D[FatherLink[x]]-1;
             if Ancestor[y] (* an upward back edge *)
             then D[SonLink[y]]:=D[SonLink[y]]+1
             else D[FatherLink[y]]:=D[FatherLink[y]]-1;
           end;
  Ancestor[x]:=false;
end;
```

Fig. 19. Algorithm to compute all D values.

An example.

In the example below (fig. 20), the algorithm Compute_D_values has been run with Root=1. For each tree edge e is indicated the final value of D[e] as well as intermediate values (overbarred) incurred during the computation. Where no number is given, the value stayed 0 throughout.

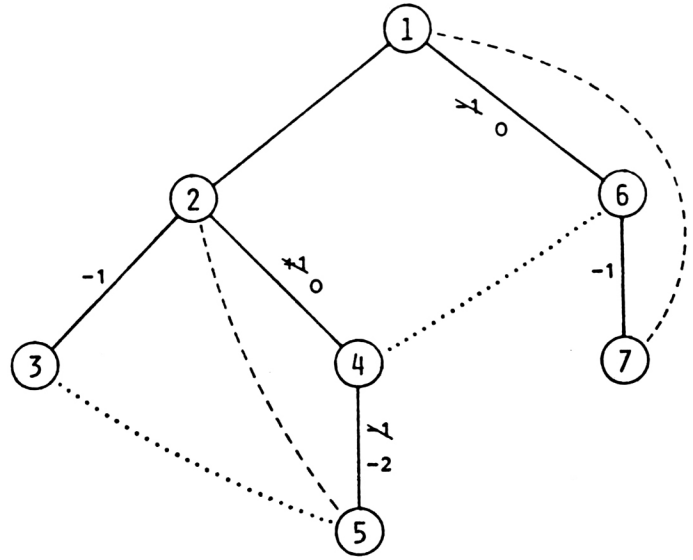

Fig. 20. D values computed in an example graph.

For instance, the first two non-tree edges that cause an update of D values are the cross edge (5,3), which makes D[(2,3)]=D[(4,5)]=-1, and the back edge (5,2), after the processing of which D[(2,4)]=+1 and D[(4,5)]=-2.

A final easy step is to use the values of D for each tree edge and the value of C(Root) (computed by CountCrossEdges) to find C(u) for all remaining nodes u. This is another tree search by the algorithm below (fig. 21), where the outermost call should be Report(Root,C(Root)).

```
procedure Report(u:Node; Cu:integer); (* Cu=C(u) *)
begin
  if Cu=0
  then writeln(u,' is a possible root');
  for all edges e=(u,v) incident on u
  do if TreeEdge[e] and (e≠FatherLink[u])
     then Report(v,Cu+D[e] (* C(v) *));
end;
```

Fig. 21. Algorithm to report all nodes u with C(u)=0.

Continuing the example above (fig. 20), C(1)=2 as evident from the figure, and we find the following C values:

$$C(1) = C(2) = C(4) = C(6) = 2$$
$$C(3) = C(7) = 1$$
$$C(5) = 0$$

Fig. 22. C values for the example graph in fig. 20.

Hence there is precisely one node, 5, which is the root of a tree with no cross edges and which may therefore be the initial node of a depth-first search associated with the given tree. Using results in (Tarjan, 1972), it is now easy to actually find such a DFS.

It is clear that all steps of the described algorithm work in linear time and space. In a more practical programming setting, the calculation of C(Root) and the computation of D values may be combined into a single tree search.

ACKNOWLEDGEMENTS

The author wishes to thank Pierre Rosenstiehl and Kurt Mehlhorn for having brought up a variant of the biconnected graph assembly problem, and his colleagues in Saarbrücken for inspiring discussions.

REFERENCES

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974), "The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Mass.

Even, S. (1979), "Graph Algorithms", Pitman, London.

Harary, F. (1969), "Graph Theory", Addison-Wesley, Reading, Mass.

Tarjan, R. E. (1972), Depth-First Search and Linear Graph Algorithms, SIAM J. Comput. 1:2, 146-160.

Valdes, J., Tarjan, R. E., and Lawler, E. L. (1979), The Recognition of Series Parallel Digraphs, in "Proc. 11th Ann. ACM Symposium on Theory of Computing", 1-12.

Whitney, H. (1932), Non-separable and planar graphs, Trans. Amer. Math. Soc. 34, 339-362.