DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS

(Working Paper)

Robert Tarjan
Computer Science Department
Stanford University
Stanford, California

Summary

The value of depth-first search or "backtracking" as a technique for solving graph problems is illustrated by two examples. An algorithm for finding the biconnected components of an undirected graph and an improved version of an algorithm for finding the strongly connected components of a directed graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Introduction

Consider a graph G, consisting of a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. The graph may either be directed (the edges are ordered pairs (v,w) of vertices; v is the tail and w is the head of the edge) or undirected (the edges are unordered pairs of vertices, also represented as (v,w)). Graphs form a suitable abstraction for problems in many areas; chemistry, electrical engineering, and sociology, for example. They are also very general structures for storing and retrieving information. Many questions may be asked about graphs; we need the fastest, most economical algorithms to give answers to these questions.

In studying graph algorithms we cannot avoid at least a few definitions. These definitions are more-or-less standard in the literature. (See Harary[1], for instance.) We shall assume a random-access computer model. If $f, f_1, \ldots, f_n$ are functions of $x_1, \ldots, x_n$, we say f is $O(f_1, \ldots, f_n)$ if $|f(x_1 \ldots x_n)| \leq k_0 + k_1|f_1(x_1 \ldots x_n)| + \ldots + k_n|f_n(x_1 \ldots x_n)|$ for all x_i and some constants $k_0 \ldots k_n$. If R and S are binary relations, R^* is the transitive closure of R, R^{-1} is the inverse of R, and $RS = \{(u,w) | \exists v((u,v) \in R \,\&\, (v,w) \in S)\}$. If $G = (\mathcal{V}, \mathcal{E})$ is a graph, a <u>path</u> $p: v \overset{*}{\Rightarrow} w$ in G is a sequence of vertices and edges leading from v to w. A path is <u>simple</u> if all its vertices are distinct. A path $p: v \overset{*}{\Rightarrow} v$ is called a <u>closed path</u>. A closed path $p: v \overset{*}{\Rightarrow} v$ is a <u>cycle</u> if all its edges are distinct and the only vertex to occur twice in p is v, which occurs exactly twice. The <u>undirected version</u> of a directed graph is the graph formed by converting each edge of the directed graph into an undirected edge and removing duplicate edges. An undirected graph is <u>connected</u> if there is a path between every pair of vertices.

A (directed rooted) <u>tree</u> T is a directed graph whose undirected version is connected, having one vertex which is the head of no edges (called the root), and such that all vertices except the root are the head of exactly one edge. The relation " (v,w) is an edge of T " is denoted by $v \rightarrow w$. The relation "There is a path from v to w in T " is denoted by $v \overset{*}{\rightarrow} w$. If $v \rightarrow w$, v is the <u>father</u> of w and w is a <u>son</u> of v. If $v \overset{*}{\rightarrow} w$, v is an <u>ancestor</u> of w and w is a <u>descendant</u> of v. If v is a vertex in a tree T, T_v is the subtree of T having as vertices all the descendants of v in T. If G is a directed graph, a tree T is a <u>spanning tree</u> of G if T is a subgraph of G and T contains all the vertices of G.

Depth-First Search

A technique which has been widely used for finding solutions to problems in combinatorial theory and artificial intelligence[6,8] but whose properties have not been widely analyzed is of great value in solving graph problems. This is backtracking, or depth-first search. Suppose G is a graph which we wish to explore. Initially all the vertices of G are unexplored. We start from some vertex of G and choose an edge to follow. Traversing the edge leads to a new vertex. We continue in this way; at each step we select an unexplored edge leading from a vertex already reached and we traverse this edge. The edge leads to some vertex, either new or already reached. Whenever we run out of edges leading from old vertices, we choose some unreached vertex, if any exists, and begin a new exploration from this point. Eventually we will traverse all the edges of G, each exactly once. Such a process is called a <u>search</u> of G.

There are many ways of searching a graph, depending upon the way in which edges to search are selected. Consider the following choice rule: when selecting an edge to traverse, always choose an edge emanating from the vertex most recently reached which still has unexplored edges. A search which uses this rule is called a <u>depth-first search</u>. The set of old vertices with possibly unexplored edges may be stored on a stack. Thus a depth-first search is very easy to program either iteratively or recursively, provided we have a suitable computer representation of a graph.

<u>Definition 1</u>: Let $G = (\mathcal{V}, \mathcal{E})$ be a graph. For each vertex $v \in \mathcal{V}$ we may construct a list containing all vertices w such that $(v,w) \in \mathcal{E}$. Such a list is called an <u>adjacency list</u> for vertex v. A set of such lists, one for each vertex in G, is called an <u>adjacency structure</u> for G.

If the graph G is undirected, each edge (v,w) is represented twice in an adjacency structure; once for v and once for w. If G is directed, each edge (v,w) is represented exactly once: vertex w appears in the adjacency list of vertex v. A single graph may have many adjacency structures; in fact, each ordering of the edges around the vertices of G gives a unique adjacency structure, and each adjacency structure corresponds to a unique ordering of the edges at each vertex. Using an adjacency structure for a graph, we can perform depth-first searches in a very efficient manner, as we shall see.

Suppose G is a connected undirected graph. A search of G imposes a direction on each edge of G given by the direction in which the edge is traversed when the search is performed. Thus G is converted into a directed graph G'. The set of edges which lead to a new vertex when traversed during the search

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.

defines a spanning tree of G'. In general, the arcs of G' which are not part of the spanning tree interconnect the paths in the tree. However, if the search is depth-first, each edge (v,w) not in the spanning tree connects vertex v with one of its ancestors w.

Definition 2: Let P be a directed graph, consisting of two disjoint sets of edges, denoted by $v \rightarrow w$ and $v \dashrightarrow w$ respectively. Suppose P satisfies the following properties:

(i) The subgraph containing the edges $v \rightarrow w$ is a spanning tree of P.

(ii) We have $\dashrightarrow \subseteq (\overset{*}{\rightarrow})^{-1}$, where " $\rightarrow$ " and " $\dashrightarrow$ " denote the relations defined by the corresponding sets of edges.

Then P is called a palm tree. The edges $v \dashrightarrow w$ are called the fronds of P.

Theorem 1: Let P be the directed graph generated by a depth-first search of a connected graph G. Then P is a palm tree. Conversely, let P be any palm tree. Then P is generated by some depth-first search of the undirected version of P.

Proof: Consider the program listed below, which carries out a depth-first search of a connected graph, starting at vertex s, using an adjacency structure of the graph to be searched. The program numbers the vertices of the graph in the order they are reached during the search and constructs the directed graph (P) generated by the search.

```
begin
   integer i;
   procedure DFS(v,u);
      begin
         NUMBER(v) := i := i+1;
         for w in the adjacency list of v do
            begin
               if w is not yet numbered then
                  begin
                     construct arc v → w in P;
                     DFS(w,v);
                  end
               else if NUMBER(w) < NUMBER(v)
                  and w ¬= u then
                     construct arc v --→ w in P;
            end;
      end;
   i := 0;
   DFS(s,0);
end;
```

Suppose $P = (\mathcal{V},\mathcal{E})$ is the directed graph generated by a depth-first search of some connected graph G, and assume that the search begins at vertex s. Examine the procedure DFS. The algorithm clearly terminates because each vertex can only be numbered once. Furthermore, each edge in the graph is examined exactly twice. Therefore the time required by the search is linear in V and E.

For any vertices v and w, let $d(v,w)$ be the length of the shortest path between v and w in G. Since G is connected, all distances are finite. Suppose that some vertex remains unnumbered by the search. Let v be an unnumbered vertex such that $d(s,v)$ is minimal. Then there is a vertex w such that w is adjacent to v and $d(s,w) < d(s,v)$. Thus w is numbered. But v will also be numbered, since it is adjacent to w. This means that all vertices are numbered during the search.

The vertex s is the head of no edge $w \rightarrow s$. Each other vertex v is the head of exactly one edge $w \rightarrow v$. The subgraph T of P defined by the edges $v \rightarrow w$ is obviously connected. Thus T is a spanning tree of P.

Each arc of the original graph is directed in at least one direction; if (v,w) does not become an arc of the spanning tree T, either $v \dashrightarrow w$ or $w \dashrightarrow v$ must be constructed, since both v and w are numbered whenever edge (v,w) is inspected and either $NUMBER(v) < NUMBER(w)$ or $NUMBER(v) > NUMBER(w)$.

The arcs $v \rightarrow w$ run from smaller numbered points to larger numbered points. The arcs $v \dashrightarrow w$ run from larger numbered points to smaller numbered points. If arc $v \dashrightarrow w$ is constructed, arc $w \rightarrow v$ is not constructed later because both v and w are numbered. If arc $w \rightarrow v$ is constructed, arc $v \dashrightarrow w$ is not later constructed, because of the test " $w \neg= u$ " in procedure DFS. Thus each edge in the original graph is directed in one and only one direction.

Let T_u be the graph defined by the arcs $v \rightarrow w$ constructed during execution of DFS(u,_). The argument above which shows that T is a spanning tree also shows that T_u is a tree rooted at u. Consider an arc $v \dashrightarrow w$. We have $NUMBER(w) < NUMBER(v)$. Thus w is numbered before edge (w,v) is inspected, and v is a vertex in T_w. This means that $w \overset{*}{\rightarrow} v$, and P is a palm tree.

Let us prove the converse part of the theorem. Suppose that P is a palm tree, with spanning tree T and undirected version G. Construct an adjacency structure of G in which all the edges of T appear before the other edges of G in the adjacency lists. Starting with the root of T, perform a depth-first search using this adjacency structure. The search will traverse the edges of T preferentially and will generate the palm tree P; it is easy to see that each edge is directed correctly. This completes the proof of the theorem.

We may state Theorem 1 non-constructively as:

Corollary 2: Let G be any undirected graph. Then G can be converted into a palm tree by directing its edges in a suitable manner.

Biconnectivity

The value of depth-first search follows from the simple structure of a palm tree. Let us consider a problem in which this structure is useful.

Definition 3: Let $G = (\mathcal{V},\mathcal{E})$ be an undirected graph. Suppose that for each triple of distinct vertices v,w,a in $\mathcal{V}$, there is a path $p: v \overset{*}{\Rightarrow} w$ such that a is not on the path P. Then G is biconnected. If, on the other hand, there is a triple of distinct vertices v,w,a in $\mathcal{V}$ such that a is on any path $p: v \overset{*}{\Rightarrow} w$, and there exists at least one such path, then a is called an articulation point of P.

Lemma 3: Let $G = (\mathcal{V},\mathcal{E})$ be an undirected graph. We may define an equivalence relation on the set of edges as follows: two edges are equivalent if and only if they belong to a common cycle. Let the distinct equivalence classes under this relation be $\mathcal{E}_i$, $1 \leq i \leq n$, and let $G_i = (\mathcal{V}_i,\mathcal{E}_i)$, where $\mathcal{V}_i$ is the set of vertices incident to the edges of $\mathcal{E}_i$; $\mathcal{V}_i = \{v | \exists w((v,w) \in \mathcal{E}_i)\}$. Then

(i) G_i is biconnected, for each $1 \leq i \leq n$.

(ii) No G_i is a proper subgraph of a biconnected subgraph of G.

(iii) Each articulation point of G occurs more than once among the $\mathcal{V}_i$, $1 \leq i \leq n$. Each non-articulation point of G occurs exactly once among the $\mathcal{V}_i$, $1 \leq i \leq n$.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.

(iv) The set $V_i \cap V_j$ contains at most one point, for any $1 \leq i,j \leq n$. Such a point of intersection is an articulation point of the graph. The subgraphs G_i of G are called the biconnected components of G^1.

Proof: An exercise for the reader.

Suppose we wish to determine the biconnected components of an undirected graph G . Common algorithms for this purpose, for instance Shirey's[10], test each vertex in turn to discover if it is an articulation point. Such algorithms require time proportional to $V \cdot E$, where V is the number of vertices and E is the number of edges of the graph. A more efficient algorithm uses depth-first search. Let **P** be a palm tree generated by a depth-first search. Suppose the vertices of **P** are numbered in the order they are reached during the search (as is done by the procedure DFS above). We shall refer to vertices by their numbers. If u is an ancestor of v in the spanning tree T of **P** , then $u < v$. For any vertex v in **P** , let LOWPT(v) be the smallest vertex in the set $\{v\} \cup \{w | v \overset{*}{\rightarrow} \dashrightarrow w\}$. The following results form the basis of an efficient algorithm for finding biconnected components.

Lemma 4: Let G be an undirected graph and let **P** be a palm tree formed by directing the edges of G . Let T be the spanning tree of **P** . Suppose $p: v \overset{*}{\Rightarrow} w$ is any path in G . Then p contains a point which is an ancestor of both v and w in T .

Proof: Let T_u with root u be the smallest subtree of T containing all vertices on the path p . If $u = v$ or $u = w$ the lemma is immediate. Otherwise, let T_{u_1} be the subtree containing v , and let T_{u_2} be the subtree containing w , such that $u \rightarrow u_1$ and $u \rightarrow u_2$. Since T_u is minimal, $u_1 \neg= u_2$. Then path p can only get from T_{u_1} to T_{u_2} by passing through vertex u , since no point in one of these trees is an ancestor of any point in the other, while both $\rightarrow$ and $\dashrightarrow$ connect only ancestors in a palm tree. Since u is an ancestor of both v and w , the lemma holds.

Lemma 5: Let G be a connected undirected graph. Let **P** be a palm tree formed by directing the edges of G , and let T be the spanning tree of **P** . Suppose a,v,w are distinct vertices of G such that $(a,v) \in T$, and suppose w is not a descendant of v in T . (That is, $\neg(v \overset{*}{\rightarrow} w)$ in T .) If $\text{LOWPT}(v) \geq a$ then a is an articulation point of **P** and removal of a disconnects v and w .

Proof: If $a \rightarrow v$ and $\text{LOWPT}(v) \geq a$, then any path from v not passing through a remains in the subtree T_v , and this subtree does not contain the point w . This gives the lemma. Figure 1 shows a graph, its LOWPT values, articulation points, and biconnected components.

The LOWPT values of all the vertices of a palm tree **P** may be calculated during a single depth-first search, since $\text{LOWPT}(v) = \min(\{\text{NUMBER}(v)\} \cup \{\text{LOWPT}(w) | v \rightarrow w\} \cup \{\text{NUMBER}(w) | v \dashrightarrow w\})$. On the basis of such a calculation, the articulation points and the biconnected components may be determined, all during one search. The biconnectivity algorithm is presented below. The program will compute the biconnected components of a graph G , starting from vertex s .

```
begin
  integer i;
  procedure BICONNECT(v,u);
    begin
      NUMBER(v) := i := i+1;
      LOWPT(v) = NUMBER(v);
      for w in the adjacency list of v do
        begin
          if w is not yet numbered then
            begin
              add (v,w) to stack of edges;
              BICONNECT(w,v);
              LOWPT(v) := min(LOWPT(v),LOWPT(w));
              if LOWPT(w) ≥ NUMBER(v) then
                begin
                  start new biconnected component;
                  for (u_1,u_2) on edge stack with
                      NUMBER(u_1) > NUMBER(v) do
                        delete (u_1,u_2) from edge stack
                          and add it to current component;
                  delete (v,w) from edge stack and add
                    it to current component;
                end;
            end
          else if NUMBER(w) < NUMBER(v) and w ¬= u then
            begin
              add (v,w) to edge stack;
              LOWPT(v) :=min(LOWPT(v),NUMBER(w));
            end;
        end;
    end;
  i := 0;
  for w a vertex do if w is not yet numbered then
    BICONNECT(w,0);
end;
```

The edges of G are placed on a stack as they are traversed; when an articulation point is found the corresponding edges are all on top of the stack. (If $(v,w) \in T$ and $\text{LOWPT1}(w) \geq \text{LOWPT1}(v)$, then the corresponding biconnected component contains the edges in $\{(u_1,u_2) | w \overset{*}{\rightarrow} u_1\} \cup \{(v,w)\}$ which are on the edge stack.) A single search on each connected component of a graph G will give us all the biconnected components of G .

Theorem 6: The biconnectivity algorithm requires $O(V,E)$ space when applied to a graph with V vertices and E edges.

Proof: The algorithm clearly requires space bounded by $k_1 V + k_2 E + k_3$, for some constants k_1 , k_2 , and k_3 . The algorithm is an elaboration of the depth-first search procedure DFS. During the search, LOWPT values are calculated and each edge is placed on the edge stack once and removed from the edge stack once. The amount of extra time required by these operations is proportional to E . Thus BICONNECT has a time bound linear in V and E .

Theorem 7: The biconnectivity algorithm correctly gives the biconnected components of any undirected graph G .

Proof: The actual depth-first search undertaken by the algorithm depends on the adjacency structure chosen to represent G ; we shall prove that the algorithm is correct for all adjacency structures. Notice first that the biconnectivity algorithm analyzes each connected component of G separately to find its biconnected components, applying one depth-first search to

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.

each connected component. Thus we need only prove that the biconnectivity algorithm works correctly on connected graphs G .

The correctness proof is by induction on the number of edges in G . Suppose G is connected and contains no edges. G either is empty or consists of a single point. The algorithm will terminate after examining G and listing no components. Thus the algorithm operates correctly in this case.

Now suppose that the algorithm works correctly on all connected graphs with E-1 or fewer edges. Consider applying the algorithm to a connected graph G with E edges. Each edge placed on the stack of edges is eventually removed and added to a component since everything on the edge stack is removed whenever the search returns to the root of the palm tree of G . Consider the situation when the first component G' is formed. Suppose that this component does not include all the edges of G . Then the vertex v currently being examined is an articulation point of the graph and separates the edges in the component from the other edges in the graph by Lemma 5.

Consider only the set of edges in the component. If BICONNECT(v,0) is executed, using the graph G' as data, the steps taken by the algorithm are the same as those taken during the analysis of the edges of G' when the data consists of the entire graph G . Since G' contains fewer edges than G , the algorithm operates correctly on G' and G' must be biconnected. If we delete the edges of G' from G , we get another subgraph G" with fewer edges than G since G' is not empty. The algorithm operates correctly on G" by the induction assumption. The behavior of the algorithm on G is simply a composite of its behavior on G' and on G" ; thus the algorithm must operate correctly on G .

Now suppose that only one component is found. We want to show that in this case G is biconnected. Suppose that G is not biconnected. Then G has an articulation point a . Consider the connected components left when a is deleted from the graph. One of them contains all the ancestors of a ; one must contain only descendants of a . The component which contains only descendants of a also contains some point v such that $(a,v) \in T$, where T is the spanning tree generated by the search. Since the component containing v has only descendants of a as vertices, it must be true that $\text{LOWPT}(v) \geq a$. But this is a contradiction, because the articulation point test would have succeeded when vertex a was examined and more than one biconnected component would be generated. This is contrary to the assumption that only one component was found. Thus G must be biconnected, and the algorithm works correctly in this case.

By induction, the biconnectivity algorithm gives the correct components when applied to any connected graph, and hence when applied to any graph.

Strong Connectivity

The biconnectivity algorithm shows how useful depth-first search can be when applied to undirected graphs. However, when a directed graph is searched in a depth-first manner, a simple palm tree structure does not result, because the direction of search on each edge is fixed. The more complicated structure which results in this case is still simple enough to prove useful in at least one application.

Definition 4: Let G be a directed graph. Suppose that for each pair of vertices v,w in G , there exist paths $p_1: v \overset{*}{\Rightarrow} w$ and $p_2: w \overset{*}{\Rightarrow} v$. Then G is said to be strongly connected.

Lemma 8: Let $G = (\mathcal{V},\mathcal{E})$ be a directed graph. We may define an equivalence relation on the set of vertices as follows: two vertices v and w are equivalent if there is a closed path $p: v \overset{*}{\Rightarrow} v$ which contains w. Let the distinct equivalence classes under this relation be $\mathcal{V}_i$, $1 \leq i \leq n$. Let $G_i = (\mathcal{V}_i,\mathcal{E}_i)$, where $\mathcal{E}_i = \{(v,w) \in \mathcal{E} | v,w \in \mathcal{V}_i\}$. Then:

(i) Each G_i is strongly connected.

(ii) No G_i is a proper subgraph of a strongly connected subgraph of G .

The subgraphs G_i are called the strongly connected components of G^1.

Proof: An exercise for the reader.

Suppose we wish to determine the strongly connected components of a directed graph. Purdom[9] and Munro[7] present virtually identical algorithms for solving this problem using depth-first search (although they do not mention this fact explicitly). Purdom claims a time bound of kV^2 for his algorithm; Munro claims $k \max(E, V \log V)$, where the graph has V vertices and E edges. Their algorithm attempts to construct a cycle by starting from a point and beginning a depth-first search. When a cycle is found, the vertices on the cycle are marked as being in the same strongly connected component and the process is repeated. The algorithm has the disadvantage that two small strongly connected components may be collapsed into a bigger one; the resultant extra work in relabelling may contribute V^2 steps using a simple approach, or $V \log V$ steps if a more sophisticated approach is used (see Munro[7]). A more careful study of what a depth-first search does to a directed graph reveals that the extensive relabelling of the Purdom-Munro algorithm may be avoided and an $O(V,E)$ algorithm may be devised.

Consider what happens when a depth-first search is performed on a directed graph G . The set of edges which lead to a new vertex when traversed during the search form a tree. The other edges fall into three classes. Some are edges running from ancestors to descendants in the tree. These edges may be ignored, because they do not affect the strongly connected components of G . Some edges run from descendants to ancestors in the tree; these we may call fronds as above. Other edges run from one subtree to another in the tree. These, we call vines. If the vertices of the tree are numbered in the order they are reached during the search, a vine (v,w) always has $\text{NUMBER}(v) > \text{NUMBER}(w)$, as the reader may easily show. We shall denote tree edges by $v \rightarrow w$ and fronds by $v \dashrightarrow w$ as before; we use $v \dashrightarrow w$ to denote vines.

Suppose G is a directed graph, to which a depth-first search algorithm is applied repeatedly until all the edges are explored. The process will create a set of trees which contain all the vertices of G , called the spanning forest F of G , and sets of fronds and vines. (Other edges are thrown away.) Suppose the vertices are numbered in the order they are reached during the search and that we refer to vertices by their number. Then we have the following results:

Lemma 9: Let v and w be vertices in G which lie in the same strongly connected component. Let u be the highest numbered common ancestor of v and w in F , a spanning forest of G generated by repeated depth-first search. Then u lies in the same strongly connected component as v and w .

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.

Proof: Without loss of generality we may assume $v \le w$. We prove the lemma by induction on v. Suppose $v = 1$. Then w is a descendant of v in F. This is true since the search explores all paths emanating from v. Thus $u = v$ and the lemma is true, trivially.

Suppose the lemma is true for all $v < k$. Let $v = k$. Consider any path $p: v \overset{*}{\Rightarrow} w$. If p contains no vertices $x < v$, then w must be a descendant of v, because p can never leave the subtree T_v, and the result holds. Otherwise, let x be the first point on path p such that $x < v$. Vertex x lies in the same strongly connected component as w, and by the induction hypothesis the highest common ancestor u' of x and w also lies in this component. The descendants of u' form an interval $(u', u'+i)$ which contains x and w and hence contains v. Thus u' is a common ancestor of v and w. Vertex u is a descendant of u', since u is the highest numbered common ancestor of v and w. Thus u must be in the same strongly connected component as v and w since there is a path $p': u' \overset{*}{\rightarrow} u \overset{*}{\rightarrow} v \overset{*}{\Rightarrow} w \overset{*}{\Rightarrow} u'$.

The lemma thus follows by induction. The proof implies that v and w actually have a common ancestor in F, which is implicit in the statement of the lemma.

Corollary 10: Let C be a strongly connected component in G. Then the vertices of C define a subtree of a tree in F, the spanning forest of G. The root of this subtree is called the root of the strongly connected component C.

The problem of finding the strongly connected components of a graph G thus reduces to the problem of finding the roots of the strongly connected components, just as the problem of finding the biconnected components of an undirected graph reduces to the problem of finding the articulation points of the graph. We can construct a simple test to determine if a vertex is the root of a strongly connected component. If v is a vertex, let LOWPT(v) be defined exactly as before: $\text{LOWPT}(v) = \min(\{w\} \cup \{w \mid v \overset{*}{\rightarrow} \dashrightarrow w\})$. Let $\text{LOWVINE}(v) = \min(\{v\}\ v\ \{w \mid v \overset{*}{\rightarrow} \dashrightarrow w\ \&\ \exists u(u \overset{*}{\rightarrow} v\ \&\ u \overset{*}{\rightarrow} w$ & u and w are in the same strongly connected component of $G)\})$.

Lemma 11: Let G be a directed graph with LOWPT and LOWVINE defined as above relative to some spanning forest F of G generated by depth-first search. Then v is the root of some strongly connected component of G if and only if $\text{LOWPT}(v) = v$ and $\text{LOWVINE}(v) = v$.

Proof: Obviously, if v is the root of a strongly connected component C of G, then $\text{LOWPT}(v) = v$ and $\text{LOWVINE}(v) = v$, since if $\text{LOWPT}(v) < v$ or $\text{LOWVINE}(v) < v$, some proper ancestor of v would be in C and v could not be the root of C.

Consider the converse. Suppose u is the root of a strongly connected component C of G, and v is a vertex in C different from u. There must be a path $p: v \overset{*}{\Rightarrow} u$. Consider the first edge on this path which leads to a vertex w not in the subtree T_v. If this edge is a frond we must have $\text{LOWPT}(v) \le w < v$. If this edge is a vine, we must have $\text{LOWVINE}(v) \le w < v$, since the highest numbered common ancestor of v and w is in C. Figure 2 shows a directed graph, its LOWPT and LOWVINE values, and its strongly connected components.

LOWPT and LOWVINE may be calculated using depth-first search. An algorithm for computing the strongly connected components of a directed graph in $O(V,E)$ time may be based on such a calculation. An implementation of such an algorithm is presented below. The points which have been reached during the search but which have not yet been placed in a component are stored on a stack. This stack is analogous to the stack of edges used by the biconnectivity algorithm.

```
begin
  integer i;
  procedure STRONGCONNECT(v);
    begin
      LOWPT(v) := LOWVINE(v) := NUMBER(v) := i := i+1;
      put v on stack of points;
      for w in the adjacency list of v do
        begin
          if w is not yet numbered then
            begin comment (v,w) is a tree arc;
              STRONGCONNECT(w);
              LOWPT(v) :=min(LOWPT(v),LOWPT(w));
              LOWVINE(v) :=min(LOWVINE(v),LOWVINE(w));
            end
          else if w is an ancestor of v do
            begin comment (v,w) is a frond;
              LOWPT(v) :=min(LOWPT(v),NUMBER(w));
            end
          else if NUMBER(w) < NUMBER(v) do
            begin comment (v,w) is a vine;
              if w is on stack of points then
                LOWVINE(v) :=min(LOWVINE(v),NUMBER(w));
            end;
        end;
      if (LOWPT(v) = NUMBER(v)) and
      (LOWVINE(v) = NUMBER(v)) then
        begin comment v is the root of a component;
          start new strongly connected component;
          while w on top of point stack satisfies
            NUMBER(w) ≥ NUMBER(v) do
              delete w from point stack and put w in
                current component;
        end;
    end;
  i := 0;
  for w a vertex if w is not yet numbered then
    STRONGCONNECT(w);
end;
```

Theorem 12: The algorithm for finding strongly connected components requires $O(V,E)$ space and time.

Proof: An exercise for the reader.

Theorem 13: The algorithm for finding strongly connected components works correctly on any directed graph G.

Proof: The calculation of LOWPT(v) performed by the algorithm is correct, since it is exactly the same as that performed by the biconnectivity algorithm. We prove by induction that the calculation of LOWVINE(v) is correct. Suppose as the induction hypothesis that for all vertices v such that v is a proper descendant of vertex k or $v < k$, LOWVINE(v) is computed correctly. This means that the test to determine if v is the root of a component is performed correctly for all such vertices v. The reader may verify that this somewhat strange induction hypothesis corresponds to considering vertices in the order they are examined for the last time during the depth-first search process.

Consider vertex $v = k$. Let $v \overset{*}{\rightarrow} w_1$ and let $w_1 \dashrightarrow w_2$ be a vine such that $w_2 < v$. The highest common ancestor u of v and w_2 is also the highest common ancestor of w_1 and w_2. If u is not in the

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.

same strongly connected component as w_2 , then there must be a strongly connected component root on the tree path $u \overset{*}{\rightarrow} w_2$. Since $w_2 < v$, this root was discovered and w_2 was removed from the stack of points and placed in a component before the edge $w_1 \dashrightarrow w_2$ is traversed during the search. Thus $w_1 \dashrightarrow w_2$ will not enter into the calculation of LOWVINE(v) . On the other hand, if u is in the same strongly connected component as w_2 , there is no component root $r \neg= u$ on the branch $u \overset{*}{\rightarrow} w_2$, and $v \dashrightarrow w_2$ will be used to calculate LOWVINE(w_2) , and also LOWVINE(v) , as desired. Thus LOWVINE(v) is calculated correctly, and by induction LOWVINE is calculated correctly for all vertices.

Since the algorithm correctly calculates LOWPT and LOWVINE , it correctly identifies the roots of the strongly connected components. If such a root u is found, the corresponding component contains all the descendants of u which are on the stack of points when u is discovered. These vertices are all on top of the stack of points, and are all put into a component by STRONGCONNECT. Thus STRONGCONNECT works correctly.

Further Applications

We have seen how the depth-first search method may be used in the construction of very efficient graph algorithms. The two algorithms presented here are in fact optimal to within a constant factor, since every edge and vertex of a graph must be examined to determine a solution to one of the problems. (Given a suitable theoretical framework, this statement may be proved rigorously.) The similarity between biconnectivity and strong connectivity revealed by the depth-first search approach is striking. The possible uses of depth-first search are very general, and are certainly not limited to the examples presented. An algorithm for finding triconnected components in O(V,E) time may be constructed by extending the biconnectivity algorithm. An algorithm for testing the planarity of a graph in O(V) time[11] is also based on depth-first search. Combining the connectivity algorithms, the planarity algorithm, and an algorithm for testing isomorphism of triconnected planar graphs,[5] we may devise an algorithm to test isomorphism of arbitrary planar graphs in O(V log V) time. Depth-first search is a powerful technique with many applications.

Acknowledgments

This research was supported by the Hertz Foundation and the National Science Foundation.

References

[1] Harary, Frank. Graph Theory. Addison-Wesley Publishing Company, Reading, Massachusetts, 1969.

[2] Hopcroft, J. and Tarjan, R. "Efficient Algorithms for Graph Manipulation." Stanford Computer Science Department Technical Report No. 207, March, 1971.

[3] Hopcroft, J. and Tarjan, R. "A V^2 Algorithm for Determining Isomorphism of Planar Graphs." Information Processing Letters 1 (1971), 32-34.

[4] Hopcroft, J. and Tarjan, R. "Planarity Testing in V log V Steps: Extended Abstract." Stanford Computer Science Department Technical Report No. 201, February, 1971.

[5] Hopcroft, J. "An N log N Algorithm for Isomorphism of Planar Triply Connected Graphs." Stanford Computer Science Department Technical Report No. 192, January, 1971.

[6] Golomb, S. W. and Baumert, L. D. "Backtrack Programming." JACM 12, 4 (Oct. 1965), 516-524.

[7] Munro, Ian "Efficient Determination of the Strongly Connected Components and Transitive Closure of a Directed Graph." Department of Computer Science, University of Toronto, 1971.

[8] Nilson, Nils J. Problem Solving Methods in Artificial Intelligence. 1970, unpublished.

[9] Purdom, Paul W. "A Transitive Closure Algorithm." Computer Science Technical Report No. 33, University of Wisconsin Computer Sciences Department, July, 1968.

[10] Shirey, R. W. "Implementation and Analysis of Efficient Graph Planarity Testing Algorithms." Ph.D. Thesis, University of Wisconsin Computer Sciences Department, June, 1969.

[11] Tarjan, R. "An Efficient Planarity Algorithm." To be published as a Stanford Computer Science Department Technical Report.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.

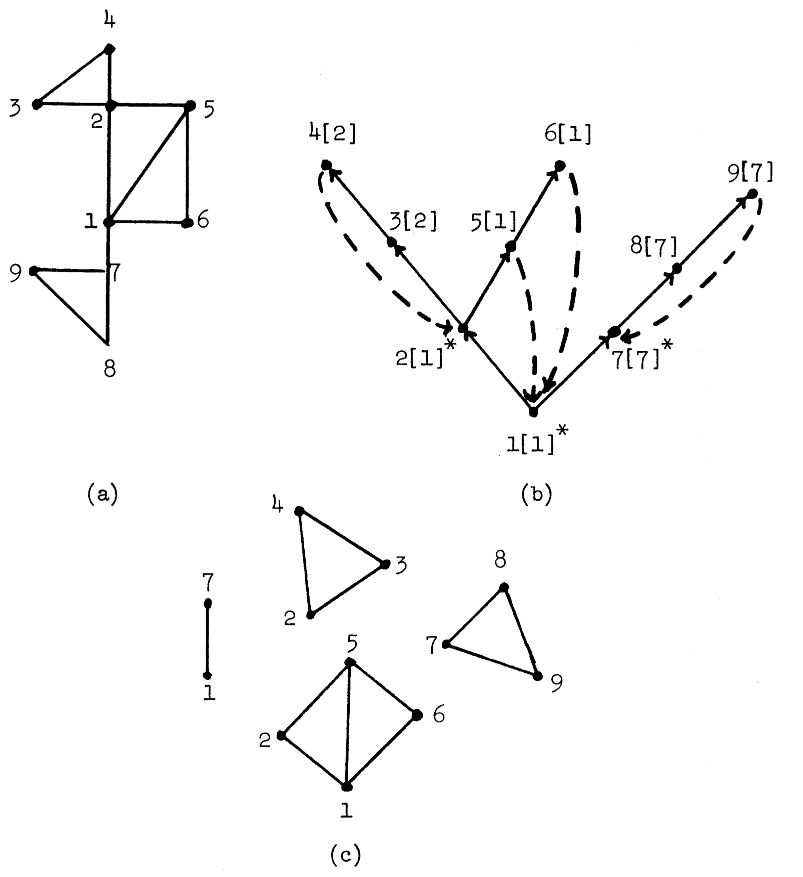

Figure 1: A graph and its biconnected components.

(a) Graph.

(b) A palm tree with LOWPT values in [ ], articulation points marked with *.

(c) Biconnected components.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.

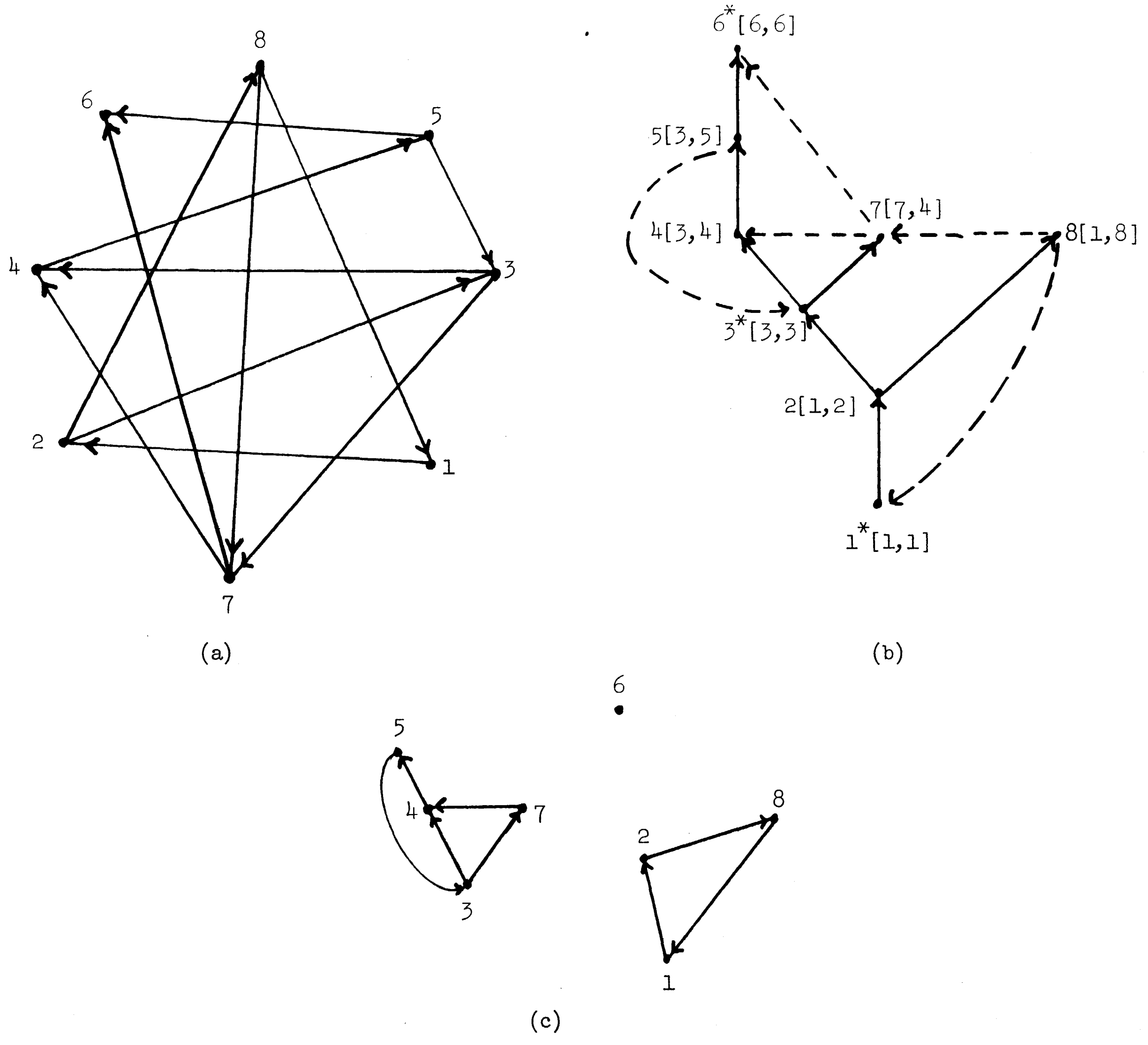

Figure 2: A graph and its strongly connected components.

(a) Graph.

(b) Corresponding graph generated by depth-first search with LOWPT, LOWVINE values in [ ], roots of components marked with *.

(c) Strongly connected components.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 18, 2009 at 08:40 from IEEE Xplore. Restrictions apply.