Fundamenta Informaticae IX (1986) 85-94
North-Holland

TWO STREAMLINED DEPTH-FIRST SEARCH ALGORITHMS

Robert Endre TARJAN

Computer Science Department, Princeton University
Princeton, NJ 08544, U.S.A.
and
AT&T Bell Laboratories, Murray Hill, NJ 07974, U.S.A.

Many linear-time graph algorithms using depth-first search have been invented. We propose simplified versions of two such algorithms, for computing a bipolar orientation or st-numbering of an undirected graph and for finding all feedback vertices of a directed graph.

1. Introduction

Depth-first search is a well-known graph exploration method that has been used to obtain efficient algorithms for a variety of graph problems [3,4,5,6,7,8,11,14,15,16,17]. Some of these algorithms, though linear-time and thus asymptotically optimal, can be simplified and thereby made easier to use and faster in practice. Our purpose in this paper is to provide two such simplified algorithms. These are for the problem of computing a bipolar orientation or an st-numbering of an undirected graph, and for finding all feedback vertices of a directed graph. In our discussion we shall assume some familiarity with the properties of depth-first search (see [1,14,15]).

2. Computing an st-Numbering

Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. G is *biconnected* if there is no vertex whose removal disconnects G. Suppose G has two distinguished vertices, s and t. A *bipolar orientation* of G is an orientation of the edges of G that produces a directed acyclic graph such that s is the unique source (vertex with no entering edges) and t is the unique sink (vertex with no exiting edges). An *st-numbering* is a numbering of the vertices of G by the integers 1 through n such that s is vertex 1, t is vertex n, and every other vertex is adjacent both to a lower-numbered and to a higher-numbered vertex.

G has a bipolar orientation if and only if it has an st-numbering, and we can compute either from the other in $O(n+m)$ time, as follows. Given a bipolar orientation, we number the vertices of G in topological order using either Knuth's algorithm [9] or depth-first search [15]. This produces an st-numbering. Given an st-numbering, we orient each edge from its lower-numbered to its higher-numbered endpoint. This produces a bipolar orientation.

The concept of an st-numbering was introduced by Lempel, Even, and Cederbaum [10], who used it in an efficient planetary-testing algorithm. Bipolar orientations and st-numberings have also been used in planar layout algorithms [12,13,19]. Lempel, Even, and Cederbaum proved that an st-numbering exists if and only if the graph $G^+ = (V, E \cup \{\{s,t\}\})$ is biconnected. Even and Tarjan [4] devised an $O(n+m)$-time algorithm for finding an st-numbering. Ebert [3] proposed a simplified version of the algorithm. Both of these methods first decompose G^+ into a collection of edge-disjoint paths and then process the paths to produce an st-numbering. We shall propose an even simpler algorithm that bypasses the path decomposition phase.

In order to develop the algorithm, we need to review some of the properties of depth-first search on undirected graphs and its use in biconnectivity testing [1,14]. Assume G^+ is connected. Suppose we carry out a depth-first search of G^+, starting at vertex s and first traversing the edge $\{s,t\}$. The search traverses every edge of G^+, orienting it in the direction along which the search advances. The resulting directed edges are of two types: *tree edges*, which define a spanning tree rooted at s and containing paths from s to every vertex, and *back edges*, each of which leads from a vertex to one of its proper ancestors in the spanning tree.

Suppose we number the vertices from 1 to n in the order they are first visited during the search. This numbering is a preorder numbering [9] of the spanning tree. We shall denote the number of a vertex v by $pre(v)$. For each vertex v, let $low(v)$ be the vertex of smallest number reachable from v by a path consisting of zero or more tree edges followed by at most one back edge. The vertex $low(v)$ is guaranteed to be an ancestor of v in the spanning tree. (See Figure 1.) The vertex numbers and the *low* values can be computed in linear time in a single depth-first search; to compute the *low* values, we use the fact that $low(v)$ is the vertex of minimum number in the set $\{v\} \cup \{low(w) \mid (v,w)$ is a tree edge$\} \cup \{w \mid (v,w)$ is a back edge$\}$. The following lemma relates *low* values to biconnectivity:

Lemma 1 [14]. G^+ is biconnected if and only if (s,t) is the only tree edge leaving s and $pre(low(w)) < pre(v)$ for every tree edge (v,w).

The st-numbering algorithm consists of two passes. The first pass is a depth-first search during which vertex numbers and *low* values are computed, as well as the parent $p(v)$ of each vertex v in the spanning tree. The first pass can if necessary check that G^+ is biconnected. The second pass constructs a list L of the vertices, such that if the vertices are numbered in the order they occur in L, an st-numbering results. The second pass is a preorder traversal of the spanning tree. During the traversal, each vertex u that is a proper ancestor of the current vertex v has a *sign* that is minus if u precedes v in L and plus if u follows v in L. Initially $L = [s,t]$ and s has sign minus. The second pass consists of repeating the following step for each vertex $v \notin \{s,t\}$ in preorder (see Figure 2):

Add a vertex. If $sign(low(v)) = plus$, insert v after $p(v)$ in L and set $sign(p(v)) = minus$; if $sign(low(v)) = minus$, insert v before $p(v)$ in L and set $sign(p(v)) = plus$.

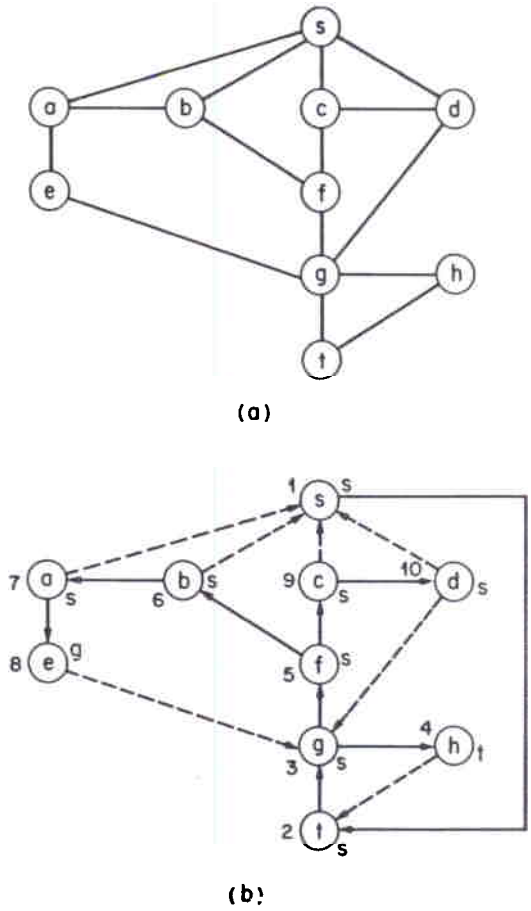

FIGURE 1.

The first pass of the *st-numbering* algorithm.

(a) An example graph G. Note that G is not biconnected, since removal of vertex g separates it.

(b) The structure imposed by a depth-first search of G^+. Vertices are numbered in preorder. The letters labeling vertices are *low* values. Tree edges are solid, back edges are dashed.

Theorem 1. The st-numbering is correct.

Proof. Consider the second pass of the algorithm. We must show that (i) the signs assigned to vertices have the claimed meaning, and (ii) if vertices are numbered in the order they occur in L, an st-numbering results.

VERTEX ADDED	LIST
	s-, t
g	s-, g, t+
h	s-, g-, h, t+
f	s-, f, g+, h, t+
b	s-, b, f+, g+, h, t+
a	s-, a, b+, f+, g+, h, t+
e	s-, a-, e, b+, f+, g+, h, t+
c	s-, a, e, b, c, f+, g+, h, t+
d	s-, a, e, b, d, c+, f+, g+, h, t+

FIGURE 2.

The list L generated by the second pass of the *st-numbering* algorithm. Irrelevant signs are omitted.

To prove (i), let $s = x_0, t = x_1, x_2, \ldots, x_l$ be the tree path from s to the vertex x_l most recently added to L, and let v with parent x_k be the next vertex to be added to L. Assume as an induction hypothesis that for all $0 \leq i < j \leq l$, $sign(x_i) = plus$ if and only if x_i follows x_j in L. The tree path from s to v is $s = x_0, t = x_1, \ldots, x_k, v$. Since $sign(x_k)$ is set to *minus* if v is inserted after x_k in L and to *plus* if v is inserted before x_k in L, the induction hypothesis holds after v is added. (For any vertex x_i for which $0 \leq i < k$, x_k and v are on the same side of x_i in L.) By induction, (i) holds.

To prove (ii), let $v \notin \{s, t\}$. If $(v, low(v))$ is a back edge, the insertion of v between $p(v)$ and $low(v)$ in L guarantees that in the numbering corresponding to L, v is adjacent to both a lower-numbered and a higher-numbered vertex. Otherwise, there must be a vertex w such that $p(w) = v$ and $low(w) = low(v)$. Lemma 1 guarantees that $low(v)$ is a proper ancestor of v, which means that $sign(low(v))$ remains constant during the time that v and w are added to L. It follows that v appears between $p(v)$ and w in the completed list L. This implies that, in the numbering corresponding to L, v is adjacent to both a lower-numbered and a higher-numbered vertex. Thus (ii) holds. □

It is obvious that the st-numbering algorithm runs in linear time. The following program, written in a variant [18] of Dijkstra's guarded command language [2], implements the algorithm. The function *st-number* is the main program, which computes and returns an st-numbering. Input to *st-number* is the vertex set V and the distinguished vertices s and t. Each vertex v is assumed to have a precomputed set $adj(v)$ of its adjacent vertices. The recursive procedure *dfs* carries out the depth-first search of the first pass. In addition to computing *pre*, *low*, and p, it constructs a list *preorder* of the vertices other than s and t in preorder for use by the second pass. In the program the empty list is denoted by "[ ]".

```
map function st-number (set V, vertex s, t);
    map pre, low, p, sign;
    list L, preorder;
    integer current;
    logical plus , minus;
    for v ∈ V → pre(v):=0 rof;
    pre(s):= current:= 1;
    preorder:= [ ];
    dfs(t);
    L = [s, t];
    plus , minus:= true , false;
    sign(S):= minus;
    for v ∈ preorder→
        if sign(low(v)) = minus→
            insert v before p(v) in L; sign(p(v)):= plus
        | sign(low(v)) = plus→
            insert v after p(v) in L; sign(p(v)):= minus
        fi
    rof;
    current:= 0;
    for v ∈ L → stnumber(v):= current:= current + 1 rof
end stnumber;
procedure dfs (vertex v);
    pre(v):= current:= current + 1;
    low(v):= v;
    for w ∈ adj(v)→
        if pre(w) = 0 →
            dfs(w);
            p(w):=v;
            preorder:= preorder & [w];
            if pre(low(w)) < pre(low(v)) → low(v):= low(w) fi
        | pre(w) ≠ 0 and pre(w) < pre(low(v)) → low(v):= w
        fi
    rof
end dfs;
```

We close this section with a few remarks. Our algorithm, in addition to being conceptually simpler than Ebert's, uses less auxiliary storage space, roughly $6n$ words instead of roughly $8n$ words. (The list L must be doubly linked to facilitate insertions both before and after given items.) Though a test for biconnectivity has been omitted, it could easily be added.

3. Finding All Feedback Vertices

Let $G = (V, E)$ be a directed graph with $|V| = n$ and $|E| = m$. Suppose $n \geqslant 2$ and G is *strongly connected*; that is, every vertex is reachable from every other. A *feedback vertex* is a vertex that lies on every cycle of G. Garey and Tarjan [5] discovered how to find all feedback vertices in $O(n+m)$ time using depth-first search. Their algorithm has four passes and is somewhat involved. We shall describe a simplified algorithm that takes only two passes.

We begin by reviewing some properties of depth-first search on directed graphs [14,15]. Suppose we carry out a depth-first search of G starting from any vertex s, numbering the vertices in the order they are *last* visited during the search. The search partitions the edges of G into four classes:

(i) *tree edges*, which define a spanning tree rooted at s containing a path from s to every vertex.

(ii) *back edges*, each of which leads from a vertex to one of its ancestors in the spanning tree.

(iii) *forward edges*, each of which leads from a vertex to one of its proper descendants in the spanning tree.

(iv) *cross edges*, each of which leads from a vertex v to a vertex w such that $number(w) < number(v)$ and v and w are unrelated in the spanning tree.

The vertex numbering is a postorder numbering [9] of the spanning tree. We shall denote the number of a vertex v by $post(v)$. An edge (v, w) has $post(v) < post(w)$ if and only if (v, w) is a back edge. The nonexistence of any edge (v, w) such that v and w are unrelated in the tree and $post(v) < post(w)$ implies the following lemma:

Lemma 2 [14]. Any path from a vertex v to a vertex w such that $post(v) \leqslant post(w)$ contains a common ancestor of v and w in the spanning tree.

We can characterize feedback vertices in terms of *post* and a second function on vertices. For any vertex v, let $high(v) = \max\{post(w) \mid w$ is reachable from v by a path consisting of zero or more non-back edges followed by at most one back edge$\}$.

Theorem 2. A vertex x is a feedback vertex if and only if

(i) for every back edge (v, w), $post(v) \leqslant post(x) \leqslant post(w)$, and

(ii) for every non-back edge (v, w) such that $high(w) \geqslant post(v)$, $post(x) \leqslant post(w)$ or $post(x) \geqslant post(v)$.

Proof. Let x be a feedback vertex. A back edge (v, w) defines a cycle consisting of (v, w) followed by the tree path from v to w; every vertex on this cycle has number between $post(v)$ and $post(w)$ (inclusive). Thus (i) holds. Let (v, w) be a non-back edge and let u be the vertex whose number is $high(w)$. There is a path p from v to u containing only u, v, w, and vertices with number less than $post(w)$. If $high(w) \geqslant post(v)$, u is an ancestor of v by Lemma 2, and the cycle consisting of p followed by the tree path from u to v contains only vertices with numbers either greater than $post(w)$ or no less than $post(v)$. Thus (ii) holds.

Conversely, let x be a non-feedback vertex such that (i) holds, let c be a simple cycle not containing x, and let z be the vertex of largest number on c. The edge on c entering z, say (y, z), is a back edge, and $post(y) \leqslant post(x) \leqslant post(z)$ by (i). There must be a nonback edge (v, w) on c such that $post(w) < post(x) < post(v)$. Consider the last such edge preceding (y, z) on c. The choice of (v, w) guarantees that the path on c from w to y contains only non-back edges; thus $high(w) \geqslant post(z) \geqslant post(v)$ and (ii) fails. □

Theorem 2 leads to an $O(n+m)$-time algorithm for finding feedback vertices. The first pass of the algorithm consists of a depth-first search that computes the number of each vertex (see Figure 3.)

The second pass is a postorder traversal of the spanning tree. It computes the *high* function and the integer *lowest*, defined to be the lowest number of a vertex entered by a back edge. Simultaneously, it manipulates a stack of candidate feedback vertices. The second pass consists of initializing $lowest = \infty$, initializing $high(v) = post(v)$ for each vertex v, and repeating the following step for each vertex v in postorder (see Figure 4):

Scan a vertex. Process each edge (v, w) out of v as follows. Replace $high(v)$ by $\max\{high(v), high(w)\}$. If (v, w) is a back edge, replace *lowest* by $\min\{lowest, post(w)\}$ and empty the stack. If (v, w) is a non-back edge such that $high(w) \geqslant post(v)$, pop from the stack every vertex with number greater than $post(w)$. After processing all edges out of v, push v onto the stack if $post(v) \leqslant lowest$.

Theorem 3. The vertices left on the stack at the end of the algorithm are exactly the feedback vertices.

Proof. Consider the second pass of the algorithm. Obviously *lowest* is computed correctly. That the computation of *high* is correct follows from the observation that for any vertex v, the value of $high(v)$ maintained by the algorithm is $post(v)$ until v is processed in postorder. Every vertex on the stack when a vertex v is processed has number less than that of v, and the numbers of stacked vertices increase from the bottom of the stack to the top. Let x be a feedback vertex. By Theorem 2 (i), x is placed on the stack; by Theorem 2 (ii), x remains on the stack until the end of the algorithm. Conversely, let x be a non-feedback vertex, and let (v, w) be the earliest-processed edge for which x fails the test of Theorem 2. If (v, w) is a back edge, then either $post(x) < post(v)$ and x will be popped from the stack when (v, w) is processed, or $post(x) > post(w)$ and x will never be pushed onto the stack. If (v, w) is a non-back edge, then x will be popped when (v, w) is processed. □

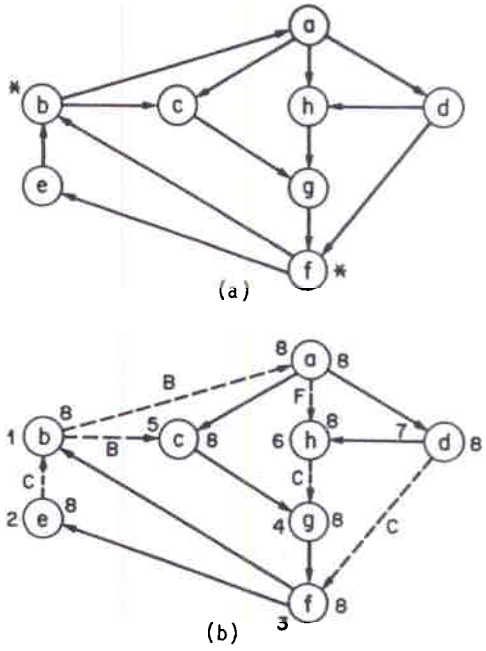

FIGURE 3.

The first pass of the feedback vertex algorithm.

(a) An example graph with the feedback vertices starred.

(b) The structure imposed by a depth-first search starting from vertex *a*. To the left of vertices are the postorder numbers assigned by the search. To the right are the *high* values. Tree edges are solid, non-tree edges are dashed. Back edges, forward edges, and cross edges are denoted by "B", "F", and "C", respectively.

VERTEX SCANNED	STACK	
b	b	
e	e,b	
f	f,b	e POPPED BY EDGE (f,b)
g	g,f,b	
c	c,g,f,b	
h	g,f,b	c POPPED BY (h,g)
d	f,b	g POPPED BY (d,f)
a	f,b	

FIGURE 4.

The behavior of the stack during the second pass of the feedback vertex algorithm. The value of *lowest* is 5.

An implementation of this algorithm appears below. The main program is the function *feedback*, which returns a list of the feedback vertices. Input to *feedback* is the vertex set V. For any vertex v, $succ(v)$ is a precomputed set of the vertices w such that (v, w) is an edge. The recursive procedure *dfs* carries out the depth-first search of the first pass. All vertex numbers are initialized to zero. When a vertex is first visited, its number is set to one; when last visited, it is numbered in postorder. The first pass constructs a list of the vertices in postorder and initializes *high*. The second pass is a loop over the postorder list.

```
list function feedback (set V);
    list postorder;
    map high;
    integer lowest, current;
    current:= 0;
    dfs(any (V));
    for v ∈ postorder→
        for w ∈ succ(v)→
            high(v):= max{high(v), high(w)};
            if post(w) > post(v)→
                lowest:= min{lowest, post(w)}; feedback:= [ ]
            | post(w) < post(v) ≤ high(w)→
                do feedback ≠ [ ] and post (top (feedback)) > post(w)→
                    pop (feedback)
                od
            fi
        rof;
        if post(v) ≤ lowest→ push v onto feedback fi
    rof
end feedback;
procedure dfs (vertex v);
    post(v):= 1;
    for w ∈ succ(v)→ if post(w) = 0→ dfs(w) fi rof;
    post(v):= high(v):= current:= current + 1
end dfs;
```

Remarks. The function **any** returns any element of a set. The function **top** returns the top element of a stack. The procedure **pop** removes the top element of a stack. The procedure **push onto** pushes a new element onto a stack. □

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[3] J. Ebert, "st-ordering the vertices of biconnected graphs," *Computing* 30 (1983), 19-33.

[4] S. Even and R. E. Tarjan, "Computing an st-numbering," *Theoretical Computer Science* 2 (1976), 339-344.

[5] M. R. Garey and R. E. Tarjan, "A linear-time algorithm for finding all feedback vertices," *Info. Proc. Letters* 7 (1978), 274-276.

[6] J. E. Hopcroft and R. E. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Comm. ACM* 16 (1973), 372-378.

[7] J. E. Hopcroft and R. E. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Comput.* 2 (1973), 135-158.

[8] J. E. Hopcroft and R. E. Tarjan, "Efficient planarity testing," *J. Assoc. Comput. Mach.* 21 (1974), 549-568.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1974.

[10] A. Lempel, S. Even, and I. Cederbaum, "An algorithm for planarity testing of graphs," *Theory of Graphs: International Symposium*, P. Rosenstiehl, ed., Gordon and Breach, New York, NY, 1967, 215-232.

[11] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flow graph," *ACM Trans. on Programming Languages and Systems* 1 (1979), 121-141.

[12] R. H. J. M. Otten and J. G. van Wijk, "Graph representations in interactive layout design," *Proc. IEEE International Symp. on Circuits and Systems* (1978), 914-918.

[13] P. Rosenstiehl and R. E. Tarjan, "Rectilinear planar layout of planar graphs and bipolar orientations," to appear.

[14] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.* 1 (1972), 146-160.

[15] R. E. Tarjan, "Finding dominators in directed graphs," *SIAM J. Comput.* 3 (1974), 62-89.

[16] R. E. Tarjan, "Testing flow graph reducibility," *J. Computer and System Sciences* 9 (1974), 355-365.

[17] R. E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Informatica* 6 (1976), 171-185.

[18] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[19] D. R. Woods, "Drawing planar graphs," Technical Report No. STAN-CS-82-943, Computer Science Dept., Stanford University, Stanford, CA, 1981.