Theoretical Computer Science 2 (1976) 339–344.
© North-Holland Publishing Company

COMPUTING AN *st*-NUMBERING*

Shimon EVEN
Computer Science Department, Technion, Haifa, Israel

Robert Endre TARJAN
Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

Communicated by Michael Paterson
Received October 1975

Abstract. Lempel, Even and Cederbaum proved the following result: Given any edge $\{s, t\}$ in a biconnected graph G with n vertices, the vertices of G can be numbered from 1 to n so that vertex s receives number 1, vertex t receives number n, and any vertex except s and t is adjacent both to a lower-numbered and to a higher-numbered vertex (we call such a numbering an st-numbering for G). They used this result in an efficient algorithm for planarity-testing. Here we provide a linear-time algorithm for computing an st-numbering for any biconnected graph. This algorithm can be combined with some new results by Booth and Lueker to provide a linear-time implementation of the Lempel–Even–Cederbaum planarity-testing algorithm.

1. Introduction

Lempel, Even and Cederbaum proved the following result [3]: Given any edge $\{s, t\}$ in a biconnected graph G with n vertices, the vertices of G may be numbered from 1 to n so that vertex s receives number 1, vertex t receives number n, and every vertex except s and t is adjacent both to a lower-numbered and to a higher-numbered vertex (we call such a numbering an *st-numbering* for G). Their proof gives an algorithm for finding such a numbering; though they state no time bound, the algorithm will run in $O(nm)$ time if the problem graph has n vertices and m edges. They used this algorithm as one step in an efficient algorithm for planarity-testing.

Here we give an $O(n + m)$-time algorithm for computing an st-numbering. The algorithm uses depth-first search. The st-numbering algorithm can be combined with a linear-time block-finding algorithm [1, 5] and with an implementation by Lueker and Booth [4] of the main part of the Lempel–Even–Cederbaum algorithm to provide a linear-time planarity test competitive with that described in [2].

* The research of the second author was supported in part by National Science Foundation Grant DCR72–03752 A02 at Stanford University and by a Miller Research Fellowship at University of California, Berkeley.

2. Terminology and preliminary results

A graph $G = (V, E)$ is a set V of $|V| = n$ *vertices* and a set E of $|E| = m$ *edges*, each of which is an unordered pair $\{v, w\}$ of distinct vertices. A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. G' is *spanning* if $V' = V$. A *path* from v_1 to v_k in G is a sequence of edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$. This path is said to *contain* edges $\{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$ and vertices $v_1, \ldots, v_k$. The path is *simple* if $v_1, v_2, \ldots, v_{k-1}, v_k$ are distinct. The path is a *cycle* if $v_1 = v_k$. We will sometimes represent a path by its sequence of vertices $v_1, v_2, \ldots, v_k$. By convention, there is a path of no edges from any vertex to itself; such a null path is not regarded as a cycle. A graph is *connected* if there is a path between any pair of vertices. The maximal connected subgraphs of a graph are vertex-disjoint and are called its *connected components*. A graph is *biconnected* if, given any three distinct vertices u, v, w, there is a path from v to w which does not contain u. The maximal biconnected subgraphs of a graph are edge-disjoint and are called its *blocks*.

A *tree* T is a connected graph which contains no cycles. In a tree there is a unique simple path between any two vertices. A *rooted tree* (T, r) is a tree with a distinguished vertex r, called the root. Given a rooted tree (T, r) and any vertices v and w, we say v is an *ancestor* of w and w is a *descendant* of v (denoted by $v \xrightarrow{*} w$) if v is contained in the path from r to w. Every vertex is an ancestor and descendant of itself. If $v \xrightarrow{*} w$ and $\{v, w\}$ is an edge of T, we say v is the *parent* of w and w is a *child* of v (denoted by $v \to w$).

A *preorder numbering* of the vertices of a rooted tree (T, r) is any numbering generated by the following algorithm.

```
algorithm PREORDER($T, r$);
  **begin**
    **procedure** SEARCH($v$);
      **begin**
        assign $v$ a number higher than all previously assigned numbers;
        **for** $w$ such that $v \to w$ **do** SEARCH($w$);
      **end** SEARCH;
    initialize all vertices to be unnumbered;
    SEARCH($r$);
  **end** PREORDER;
```

A rooted tree (T, r) is a *depth-first spanning tree* of a graph G if

(a) T is a spanning subgraph of G; and

(b) each edge $\{v, w\}$ in G but not in T satisfies $v \xrightarrow{*} w$ or $w \xrightarrow{*} v$ in T.

The edges in T are called the *tree edges* of G; the edges $\{v, w\}$ in G but not in T are called the *cycle edges* of G, denoted by $v \dashrightarrow w$.

Given any graph G, a depth-first spanning tree (T, r) of G can be found in $O(n+m)$ time by using depth-first search [5]. This search can also be used to number the vertices of T from 1 to n in preorder. Henceforth assume that (T, r) is a depth-first spanning tree of G with vertices numbered from 1 to n in preorder, and that vertices are identified by their preorder number.

For any vertex v, let

$$L(v) = \min(\{v\} \cup \{u \mid \exists w \text{ such that } v \overset{*}{\rightarrow} w \text{ and } w \dashrightarrow u\}).$$

Lemma 1 (Tarjan [5]). *If G is biconnected and $v \rightarrow w$, then $v \neq 1$ implies $L(w) < v$, and $v = 1$ implies $L(w) = v = 1$.*

The values $L(v)$ for all vertices v may be computed in $O(n+m)$ time during the search which constructs (T, r).

3. Computing an *st*-numbering

Given a biconnected graph G and a distinguished edge $\{s, t\}$, we wish to construct an *st*-numbering for G. The *st*-numbering algorithm consists of three parts. The first part is a depth-first search which constructs a depth-first spanning tree (T, t) of G so that the first edge of the search is $\{t, s\}$; thus $t \rightarrow s$ in T and since G is biconnected, there is no other tree edge emanating from t. This search also numbers the vertices of T from 1 to n in preorder and computes the values $L(v)$. The information generated by this search is used by the other two parts of the algorithm.

The second part of the algorithm is a pathfinding procedure PATHFINDER(v) to be used in the following way: Initially only vertices s, t and edge $\{s, t\}$ are marked *old*. The initial call PATHFINDER(s) finds a simple path from s to t not containing $\{s, t\}$, and marks edges and vertices on this path *old*. Each successive call PATHFINDER(v) finds a simple path of new edges from old vertex v to some distinct old vertex w, marks edges and vertices on the path old, and returns the path. The pathfinder procedure presented here is a simpler version of the one presented in [1].

```
procedure PATHFINDER($v$);
  begin
    if there is a new cycle edge $\{v, w\}$ with $w \overset{*}{\rightarrow} v$ then
(a)   begin
        mark $\{v, w\}$ old;
        let path be $\{v, w\}$;
      end
    else if there is a new tree edge $v \rightarrow w$ then
```

```
(b)    begin
         mark $\{v, w\}$ old;
         initialize path to be $\{v, w\}$;
         while $w$ is new do
           begin
             find a (new) edge $\{w, x\}$ with $x = L(w)$ or $L(x) = L(w)$
             mark $w$ and $\{w, x\}$ old;
             add $\{w, x\}$ to path;
             $w := x$;
       end end
     else if there is a new cycle edge $\{v, w\}$ with $v \overset{*}{\rightarrow} w$ then
(c)    begin
         mark $\{v, w\}$ old;
         initialize path to be $\{v, w\}$;
         while $w$ is new do
           begin
             find the (new) edge $\{w, x\}$ with $x \rightarrow w$;
             mark $w$ and $\{w, x\}$ old;
             add $\{w, x\}$ to path;
             $w := x$;
       end end
     else
(d)    let path be null path;
  end PATHFINDER;
```

Lemma 2. *Suppose vertices s, t and edge $\{s, t\}$ are initially marked old. An initial call PATHFINDER(s) will return a simple path from s to t not containing $\{s, t\}$. A successive call PATHFINDER(v) with v old will return a simple path (of edges new before the call) from v to some vertex w old before the call, if there are any edges $\{v, w\}$ new before the call. (Otherwise PATHFINDER(v) returns the null path.)*

Proof. It is easy to prove by induction on the number of PATHFINDER calls that, at the beginning of any PATHFINDER call, if some vertex w is old, then all vertices and edges on the tree path from t to w are old. Given this fact, we can prove the lemma by considering statements (a)–(d), one of which is executed each time PATHFINDER is called. If choice (a) or (d) is made, PATHFINDER obviously performs according to the statement of the lemma. Consider choice (b). By Lemma 1, $L(w) < v$, where $\{v, w\}$ is the first edge on the path. Thus statement (b) selects some path $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$, where $v_i \rightarrow v_{i+1}$ for $1 \leq i < k$, $v_{k-1} - v_k$, $v_k \overset{*}{\rightarrow} v_{k-1}$, and $v_k = L(v_2) < v_1$. Hence the selected path is simple. Consider choice (c). Since choice (b) is not made, all vertices x such that $v \rightarrow x$ are old when choice (c) is made. Thus the selected path terminates at some descendant of v (not v) and is simple. □

The third part of the *st*-numbering method uses procedure PATHFINDER to compute an *st*-numbering. The method keeps a stack of all old vertices. Initially the stack contains s on top of t. The top vertex on the stack, say v, is deleted and PATHFINDER(v) is called. If PATHFINDER returns a path $\{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$, then $v_{k-1}, v_{k-2}, \ldots, v_2$ and $v_1 = v$ are added to the top of the stack. If PATHFINDER returns the null path, v is assigned the next available number (and *not* put back on the stack). The process is repeated until the stack is empty. This algorithm appears below in ALGOL-like notation.

```
algorithm STNUMBER;
  begin
    mark $s$, $t$, and $\{s, t\}$ old, all other vertices and edges new;
    initialize stack to contain $s$ on top of $t$;
    $i := 0$;
    while stack is not empty do
      begin
        let $v$ be top vertex on stack;
        delete $v$ from stack;
        let $\{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$ be path found by PATHFINDER($v$);
        if path not null then add $v_{k-1}, \ldots, v_1$ ($v_1 = v$ on top) to stack
        else $number(v) := i := i + 1$;
  end end STNUMBER;
```

Lemma 3. *Algorithm STNUMBER correctly computes an st-numbering of a biconnected graph G.*

Proof. First we make some observations about STNUMBER. No vertex v ever appears in two or more places on *stack* at the same time. Once a vertex v is placed on *stack*, nothing under v receives a number until v does. No vertex is permanently deleted from the stack until all its incident edges become old. We use these facts to prove that each vertex $u \neq s, t$ is placed on *stack* before t is deleted, and that the numbering computed by STNUMBER is an *st*-numbering.

Let $u \neq s, t$. Since G is biconnected there is a path $s = w_1, w_2, \ldots, w_k = u$ from s to u which does not contain t. Let w_j be the vertex with largest j which is placed on *stack* before t is deleted. Suppose $w_j \neq w_k$. Then w_j must be permanently deleted from *stack* before t is deleted. But w_j cannot be deleted until edge $\{w_j, w_{j+1}\}$ is old, and this in turn requires that w_{j+1} be placed on *stack*. This contradiction implies $w_j = w_k$, and hence $u = w_k$ is placed on *stack* before t is deleted.

Since all vertices are eventually placed on *stack*, all vertices receive a number. Since s is the first vertex permanently deleted from *stack* and t is the last, s receives number 1 and t receives number n. Consider any vertex $u \neq s, t$. To place u on *stack* (the first time), STNUMBER (using PATHFINDER) must find a simple path $v_1, v_2, \ldots, v_k$ with v_1, v_k old, $v_2, \ldots, v_{k-1}$ new, and $u = v_i$ for some $2 \leq i \leq k-1$. When u is placed on *stack*, v_{i+1} is below u on *stack*, and v_{i-1} is then placed on top of u on *stack*. Thus v_{i+1} receives a larger number than u and v_{i-1} a smaller number. It follows that the numbering is an *st*-numbering. $\square$

The running time of the *st*-numbering algorithm is $O(n+m)$ for the depth-first search plus the time required by STNUMBER. The time required by STNUMBER is dominated by the time spent in PATHFINDER calls. PATHFINDER can easily be implemented so that a call requires time proportional to one plus the number of edges on the path found. This requires only that, for each vertex v, the following items be kept: a list of cycle edges $\{v, w\}$ with $v \overset{*}{\rightarrow} w$; a list of cycle edges $\{v, w\}$ with $w \overset{*}{\rightarrow} v$; a list of tree edges $v \rightarrow w$; the tree edge $u \rightarrow v$ (if any); and an edge $\{v, w\}$ such that $w = L(v)$ or $L(w) = L(v)$. All these items can be constructed during the depth-first search, and their storage requires linear space. Thus the total time for all calls on PATHFINDER is $O(n+m)$ (each edge occurs in exactly one path), and the time to find an *st*-numbering is $O(n+m)$.

4. An application

The last section has presented an $O(n+m)$ algorithm to find an *st*-numbering in a biconnected graph. The algorithm uses depth-first search. The crux of the result is the relationship between depth-first search and biconnectivity (Lemma 1), which is used in a procedure for partitioning a biconnected graph into simple paths. This pathfinding procedure is then used to find the desired *st*-numbering. The result gives an alternate proof of the Lempel–Even–Cederbaum result that an *st*-numbering *exists* for any biconnected graph.

We can combine the *st*-numbering algorithm with an $O(n+m)$ block-finding algorithm [1, 5] and with an efficient implementation by Lueker and Booth [4] of the main part of the Lempel–Even–Cederbaum planarity algorithm [3] to give an $O(n)$ implementation of the Lempel–Even–Cederbaum planarity test (see [4]). This implementation should be comparable in speed and complexity with the $O(n)$ algorithm described in [2].

References

[1] J. Hopcroft and R. Tarjan, Algorithm 147: efficient algorithms for graph manipulation, *Comm. ACM* **16** (1973) 372–378.

[2] J. Hopcroft and R. Tarjan, Efficient planarity testing, *J. ACM* 21 (1974) 549–568.

[3] A. Lempel, S. Even and I. Cederbaum, An algorithm for planarity testing of graphs, in: P. Rosenstiehl (ed.), *Theory of Graphs: International Symposium, July* 1966 (Gordon and Breach, New York, 1967) 215–232.

[4] G. S. Lueker and K. S. Booth, Linear algorithms to recognize interval graphs and test for the consecutive ones properly, *Proc. Seventh Annual ACM Symp. on Theory of Computing* (1975) 255–265.

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