Eager *st*-Ordering

Ulrik Brandes

Department of Computer & Information Science, University of Konstanz
ulrik.brandes@uni-konstanz.de

Abstract. Given a biconnected graph $G = (V, E)$ with edge $\{s, t\} \in E$, an *st*-ordering is an ordering $v_1, \ldots, v_n$ of V such that $s = v_1$, $t = v_n$, and every other vertex has both a higher-numbered and a lower-numbered neighbor. Previous linear-time *st*-ordering algorithms are based on a preprocessing step in which depth-first search is used to compute lowpoints. The actual ordering is determined only in a second pass over the graph. We present a new, incremental algorithm that does not require lowpoint information and, throughout a single depth-first traversal, maintains an *st*-ordering of the biconnected component of $\{s, t\}$ in the traversed subgraph.

1 Introduction

The *st*-ordering of vertices in an undirected graph is a fundamental tool for many graph algorithms, e.g. in planarity testing, graph drawing, or message routing. It is closely related to other important concepts such as biconnectivity, ear decompositions or bipolar orientations.

The first linear-time algorithm for *st*-ordering the vertices of a biconnected graph is due to Even and Tarjan [2, 3]. Ebert [1] presents a slightly simpler algorithm, which is further simplified by Tarjan [7]. All these algorithms, however, preprocess the graph using depth-first search, essentially to compute lowpoints which in turn determine an (implicit) open ear decomposition. A second traversal is required to compute the actual *st*-ordering.

We present a new algorithm that avoids the computation of lowpoints and thus requires only a single pass over the graph. It appears to be more intuitive, explicitly computes an open ear decomposition and a bipolar orientation on the fly, and it is robust against application to non-biconnected graphs. Most notably, it can be stopped after any edge traversal and will return an *st*-ordering of the biconnected component containing $\{s, t\}$ in what has been traversed of the graph until then. The algorithm can thus be utilized in lazy evaluation, for instance when only the ordering of few vertices is required, and on implicitly represented graphs that are costly to traverse more than once.

The paper is organized as follows. In Sect. 2 we recall basic definitions and correspondences between biconnectivity, *st*-orderings and related concepts. Section 3 is a brief review of depth-first search and lowpoint computation. The new algorithm is developed in Sect. 4 and discussed in Sect. 5.

R. Möhring and R. Raman (Eds.): ESA 2002, LNCS 2461, pp. 247–256, 2002.

2 Preliminaries

We consider only undirected and simple graphs $G = (V, E)$. A (simple) *path* $P = (v_0, e_1, v_1, \ldots, e_k, v_k)$ in G is an alternating sequence of vertices $V(P) = \{v_0, \ldots, v_k\} \subseteq V$ and edges $E(P) = \{e_1, \ldots, e_k\} \subseteq E$ such that $\{v_{i-1}, v_i\} = e_i$, $1 \leq i \leq k$, and $v_i = v_j$ implies $i = j$ or $\{i, j\} = \{0, k\}$. The *length* of P is k. A path is called *closed* if $v_0 = v_k$, and *open* otherwise.

A graph G is *connected* if every pair of vertices is linked by a path, and it is *biconnected* if it remains connected after any vertex is removed from G.

We are interested in ordering the vertices of a biconnected graph in a way that guarantees forward and backward connectedness. Determining such an ordering is an essential preprocessing step in many applications including planarity testing, routing, and graph drawing.

Definition 1 (*st*-ordering [5]). *Let $G = (V, E)$ be a biconnected graph and $s \neq t \in V$. An ordering $s = v_1, v_2, \ldots, v_n = t$ of the vertices of G is called an* st*-*ordering*, if for all vertices v_j, $1 < j < n$, there exist $1 \leq i < j < k \leq n$ such that $\{v_i, v_j\}, \{v_j, v_k\} \in E$.*

Lemma 1 ([5]). *A graph $G = (V, E)$ is biconnected if and only if, for each edge $\{s, t\} \in E$, it has an st-ordering.*

Several linear-time algorithms for computing *st*-orderings of biconnected graphs are available [2, 1, 7]. All of these are based on a partition of the graph into oriented paths.

An *orientation* assigns a direction to each edge in a set of edges. An *st-orientation* (also called a *bipolar orientation*) of a graph G is an orientation such that the resulting directed graph is acyclic and s and t are the only source and sink, respectively. The following lemma is folklore.

Lemma 2. *A graph $G = (V, E)$ has an st-orientation if and only if it has an st-ordering. These can be transformed into each other in linear time.*

Proof. An *st*-ordering is obtained from an *st*-orientation by topological ordering, and an *st*-orientation is obtained from an *st*-ordering by orienting edges from lower-numbered to higher-numbered vertices. □

A sequence $D = (P_0, \ldots, P_r)$ of (open) paths inducing graphs $G_i = (V_i, E_i)$ with $V_i = \bigcup_{j=0}^{i} V(P_j)$ and $E_i = \bigcup_{j=0}^{i} E(P_j)$, $0 \leq i \leq r$, is called an (*open*) *ear decomposition*, if $E(P_0), \ldots, E(P_r)$ is a partititon of E and for each $P_i = (v_0, e_1, v_1, \ldots, e_k, v_k)$, $1 \leq i \leq r$, we have $\{v_0, v_k\} \subseteq V_{i-1}$ and $\{v_1, \ldots, v_{k-1}\} \cap V_{i-1} = \emptyset$. An ear decomposition *starts* with edge $\{s, t\} \in E$, if $P_0 = (s, \{s, t\}, t)$.

Lemma 3 ([8]). *A graph $G = (V, E)$ is biconnected if and only if, for each edge $\{s, t\} \in E$, it has an open ear decomposition starting with $\{s, t\}$.*

Note that, given an open ear decomposition $P_0, P_1, \ldots, P_r$ starting with edge $\{s,t\}$, it is straightforward to construct an *st*-orientation. Simply orient P_0 from s to t, and $P_i = (u, \ldots, w)$, $1 \leq i \leq r$, from u to w (from w to u) if u lies before (after) w in the partial ordering induced by $P_0, \ldots, P_{i-1}$. Since the orientation of an ear conforms to the order of its endpoints, no cycles are introduced, and s and t are the only source and sink.

3 Depth-First Search and Biconnectivity

Starting from a *root* vertex s, a *depth-first search* (DFS) of an undirected graph $G = (V, E)$ traverses all edges of the graph, where the next edge is chosen to be incident to the most recently visited vertex that has an untraversed edge. An edge $\{v, w\}$ traversed from v to w is called *tree edge*, denoted by $v \rightarrow w$, if w is encountered for the first time when traversing $\{v, w\}$, and it is called *back edge*, denoted by $v \hookrightarrow w$, otherwise. For convenience we use $v \rightarrow w$ or $v \hookrightarrow w$ to denote the respective edge as well as the fact that there is such an edge between v and w, or the path $(v, \{v, w\}, w)$. We denote by $v \xrightarrow{*} w$ a (possibly empty) path of tree edges traversed in the corresponding direction. Note that the tree edges form a spanning *DFS tree* $T = T(G)$ rooted at s, i.e. $s \xrightarrow{*} v$ for all $v \in V$ and $v \hookrightarrow w$ implies $w \xrightarrow{*} v$. For $v \in V$ let $T(v)$ be the subtree induced by all $w \in V$ with $v \xrightarrow{*} w$. We will use the graph in Fig. 1 as our running example.

DFS is the basis of many efficient graph algorithms [6], which often make use of the following notion. The *lowpoint* of a vertex $u \in V$ is the vertex w closest to s in $T(G)$ with $w = u$ or $u \xrightarrow{*} v \hookrightarrow w$. If no such path exists, u is its own lowpoint. Lowpoints are important mostly for the following reason.

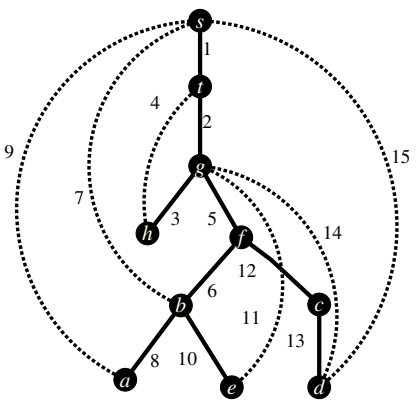


Fig. 1. Running example with edges numbered in DFS traversal order, tree edges depicted solid, and back edges dashed (redrawn from [7])

Lemma 4 ([6]). *A graph $G = (V, E)$ is biconnected if and only if, in a DFS tree $T(G)$, only the root is its own lowpoint and there is at most one tree edge $s \to t$ such that s is the lowpoint of t (in this case, s is the root).*

Previous linear-time *st*-ordering algorithms first construct a DFS tree and simultaneously compute lowpoints for all vertices. In a second traversal of the graph they use this information to determine an *st*-ordering. A new biconnectivity algorithm by Gabow [4] that requires only one pass over the graph and in particular does not require lowpoints raised the question whether we can *st*-order the vertices of a biconnected graph in a similar manner.

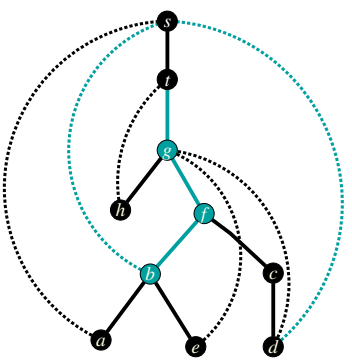
4 An Eager Algorithm

We present a new linear-time algorithm for *st*-ordering the vertices of a biconnected graph. It is eager in the sense that it maintains, during a depth-first search, an ordering of the maximum traversed subgraph for which such ordering is possible without the potential need to modifiy it later on. It is introduced via three preliminary steps that indicate how the algorithm also computes on the fly an open ear decompostion and an *st*-orientation. Pseudo-code for the complete algorithm is given at the end of this section.

4.1 Open Ear Decomposition

Let $G = (V, E)$ be a biconnected graph with $\{s, t\} \in E$, and let T be a DFS tree of G with root s and $s \to t$. We define an open ear decomposition $D(T) = (P_0, \ldots, P_r)$ using local information only. In particular, we do not make use of lowpoint values.

Let $P_0 = (s, \{s, t\}, t)$ and assume that we have defined $P_0, \ldots, P_i$, $i \geq 0$. If there is a back edge left that is not in E_i, we define P_{i+1} as follows. Let



P_0 : $s \to t$
P_1 : $t \to g \to f \to b \hookrightarrow s$
P_2 : $g \to h \hookrightarrow t$
P_3 : $b \to a \hookrightarrow s$
P_4 : $b \to e \hookrightarrow g$
P_5 : $f \to c \to d \hookrightarrow g$
P_6 : $d \hookrightarrow s$ (trivial ear)

Fig. 2. An ear decomposition $D(T)$ obtained from DFS tree T

$v, w, x \in V$ such that $w, x \in V_i$, $v \hookrightarrow w \notin E_i$, $w \rightarrow x$, and $x \overset{*}{\rightarrow} v$ (see Fig. 2). Using the last vertex u on the tree path from x to v with $u \in V_i$ (potentially v itself), we set $P_{i+1} = u \overset{*}{\rightarrow} v \hookrightarrow w$. Since $w \rightarrow x \overset{*}{\rightarrow} u$, P_{i+1} is open. It is called *ear of* $v \hookrightarrow w$, and *trivial* if $u = v$.

Theorem 1. *$D(T)$ is an open ear decomposition starting with $\{s, t\}$.*

Proof. Clearly, $D(T) = (P_0, \ldots, P_r)$ is a sequence of edge-disjoint open paths. It remains to show that they cover the entire graph, i.e. $V_r = V$ and $E_r = E$.

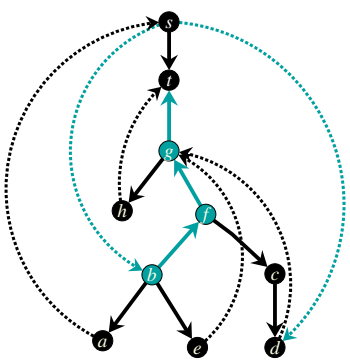
First assume there is an uncovered vertex and choose $u \notin V_r$ such that $s \overset{*}{\rightarrow} u$ has minimum length. Let w be the lowpoint of u. Since G is biconnected and $u \neq s, t$, Lemma 4 implies that there exist $x, v \in V$ with $w \rightarrow x \overset{*}{\rightarrow} u \overset{*}{\rightarrow} v \hookrightarrow w$ and $x \neq u$. It follows that $w, x \in V_r$ by minimality of u, so that the decomposition is incomplete, since $v \hookrightarrow w$ satisfies all conditions for another ear.

Since all vertices are covered, all tree edges are covered by construction. Finally, an uncovered back edge satisfies all conditions to define another (trivial) ear. □

4.2 *st*-Orientation

We next refine the above definition of an open ear decomposition to obtain an *st*-orientation. We say that a back edge $v \hookrightarrow w$, and likewise its ear, *depends* on the unique tree edge $w \rightarrow x$ for which $x \overset{*}{\rightarrow} v$. Ears in $D(T)$ are oriented in their sequential order: P_0 is oriented from s to t, whereas P_i, $0 < i \leq r$, is oriented according to the tree edge it depends on. See Fig. 3 for an example.

The following lemma shows that orienting back edges and their ears parallel to the tree edge they depend on nicely propagates into subtrees. If $\{v, w\} \in E_i$, $0 \leq i \leq r$, is oriented from v to w let $v \prec_i w$, $0 \leq i \leq r$.



$P_1 = t \overset{*}{\rightarrow} b \hookrightarrow s$ depends on $s \rightarrow t$
$P_2 = g \overset{*}{\rightarrow} h \hookrightarrow t$ depends on $t \rightarrow g$
$P_3 = b \overset{*}{\rightarrow} a \hookrightarrow s$ depends on $s \rightarrow t$
$P_4 = b \overset{*}{\rightarrow} e \hookrightarrow g$ depends on $g \rightarrow f$
$P_5 = f \overset{*}{\rightarrow} d \hookrightarrow g$ depends on $g \rightarrow f$
$P_6 = d \overset{*}{\rightarrow} d \hookrightarrow s$ depends on $s \rightarrow t$

Fig. 3. Orientation of ears in $D(T)$

Lemma 5. *For all $0 \leq i \leq r$, the above orientation of $P_0, \ldots, P_i$ yields an st-orientation of G_i, and $\prec_i$ is a partial order satisfying: If $w \to x \in E_i$ and $w \prec_i x$ ($x \prec_i w$), then $w \prec_i v$ ($v \prec_i w$) for all $v \in T(x) \cap V_i$.*

Proof. The proof is by induction over the sequence $D(T)$. The invariant clearly holds for P_0. Assume it holds for some $i < r$ and let P_{i+1} be the ear of $v \hookrightarrow w$. Let $w \to x \in E_i$ be the tree edge that $v \hookrightarrow w$ depends on and assume it is oriented from w to x (the other case is symmetric). The last vertex $u \in V_i$ on $x \overset{*}{\to} v$ satisfies $w \prec_i u$. All vertices of P_{i+1} except w are in $T(x)$, and since P_{i+1} is oriented like $w \to x$, the invariant is maintained. □

Corollary 1. *The above orientation of $D(T)$ yields an st-orientation of G.*

4.3 *st*-Ordering

We finally show how to maintain incrementally an ordering of V_i during the construction of $D(T)$. Starting with the trivial *st*-ordering of P_0, let $P_i = u \overset{*}{\to} v \hookrightarrow w$, $0 < i \leq r$, be the ear of $v \hookrightarrow w$. If P_i is oriented from u to w (w to u), insert the sequence of inner vertices $V(P_i) \setminus \{u, w\}$ of P_i in the order given by the orientation of P_i immediately after (before) u.

Lemma 6. *For all $0 \leq i \leq r$, the ordering of V_i is a linear extension of $\prec_i$.*

Proof. The proof is again by induction over the sequence $D(T)$. The invariant clearly holds for P_0. Assume that it holds for some $i < r$ and that $P_{i+1} = u \overset{*}{\to} v \hookrightarrow w$ depends on $w \to x$. If $w \prec_i x$ (the other case is symmetric), the inner vertices of P_{i+1} are inserted immediately before u. By Lemma 5 and our invariant, u is after w, so that the ordering is also a linear extension of $\prec_{i+1}$. □

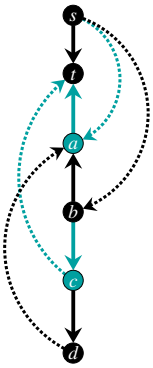
Corollary 2. *The above ordering yields an st-ordering of G.*

Note that inserting the inner vertices of an ear $P = u \overset{*}{\to} v \hookrightarrow w$ next to its *destination* w rather than its *origin* u may result in end vertices of another ear being in the wrong relative order during a later stage of the algorithm. An example of this kind is shown in Fig. 4.

4.4 Algorithm and Implementation Details

It remains to show how the above steps can be carried out in linear time. We base our algorithm on depth-first search, but unlike previous algorithms, DFS is not used for preprocessing but rather as the template of the algorithm.

The algorithm is given in Alg. 1, where code for DFS-tree management is implicit. Each time DFS traverses a back edge $v \hookrightarrow w$, we check whether the tree edge $w \to x$ it depends on is already oriented (note that x is the current child of w on the DFS path). If $w \to x$ is oriented, the ear of $v \hookrightarrow w$ is oriented in the same direction and inserted into the ordering. For each tree edge in a newly



insertion at destination	insertion at origin
$\underline{s}, \underline{t}$	$\underline{s}, \underline{t}$
$s, \underline{a}, t$	$s, \underline{a}, t$
$s, \underline{b}, a, t$	$s, \underline{b}, a, t$
$s, b, a, \underline{c}, t$	$s, b, \underline{c}, a, t$
$c \not\prec a$	$s, b, c, \underline{d}, a, t$

Fig. 4. Example showing that it is important to insert an ear next to its origin in the tree rather than the destination of its defining back edge

Algorithm 1: Eager *st*-ordering

Input: graph $G = (V, E)$, edge $\{s, t\} \in E$
Output: list L of vertices in biconnected component of $\{s, t\}$ (in *st*-order)

```
process_ears(tree edge w → x) begin
    foreach v ↪ w depending on w → x do
        determine u ∈ L on x →* v closest to v;
        set P to u →* v ↪ w;
        if w → x oriented from w to x (resp. x to w) then
            orient P from w to u (resp. u to w);
            insert inner vertices of P into L right before (resp. after) u;
        foreach tree edge w' → x' of P do process_ears(w' → x');
    clear dependencies on w → x;
end
dfs(vertex v) begin
    foreach neighbor w of v do
        if v → w then dfs(w);
        if v ↪ w then
            let x be the current child of w;
            make v ↪ w depend on w → x;
            if x ∈ L then process_ears(w → x);
end
begin
    initialize L with s → t;
    dfs(t);
end
```

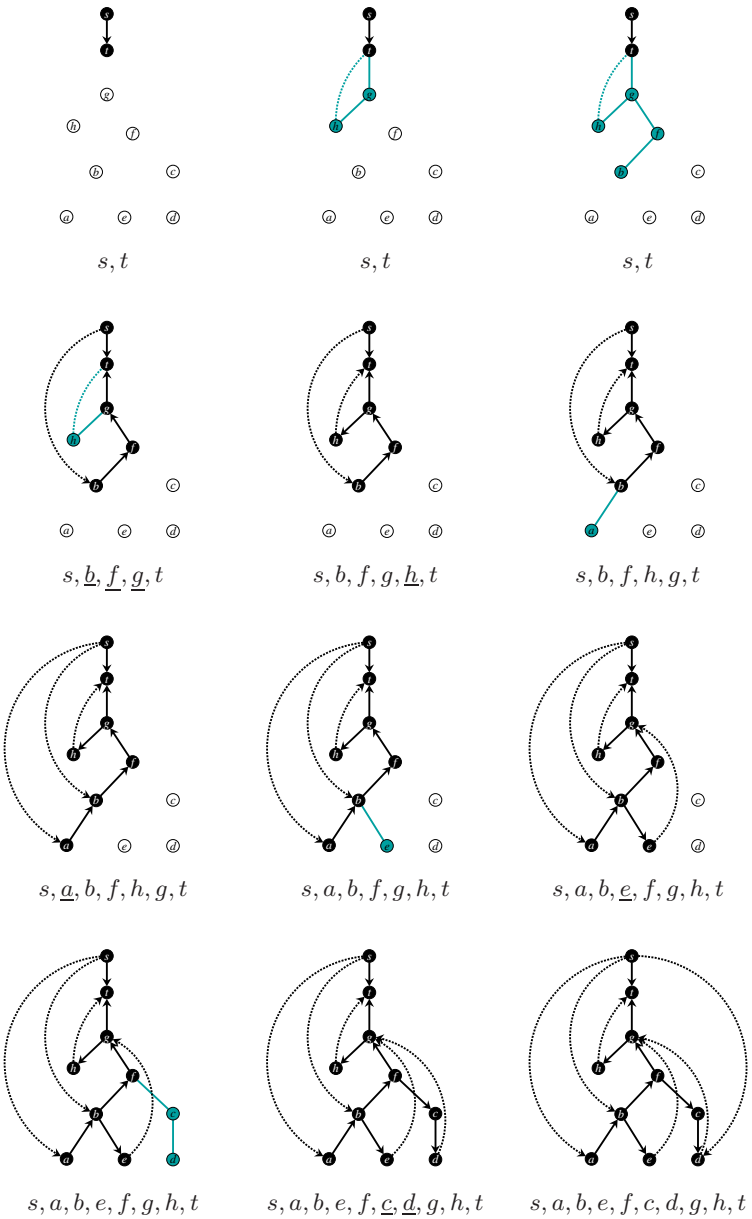


Fig. 5. Intermediate steps of the algorithm

oriented ear, we recursively process the ears of not yet oriented back edges that depend on it.

After each traversal of an edge we are therefore left with an *st*-ordering of the biconnected component of $\{s,t\}$ in the traversed subgraph. Consequently, the algorithm is robust against application to non-biconnected graphs, and the input graph is biconnected if and only if the ordering returned contains all vertices.

Theorem 2. *Algorithm 1 computes an st-ordering of the biconnected component containing* $\{s,t\}$ *in linear time.*

Proof. Given the discussion above, it is sufficient to show that the algorithm determines the entire open ear decomposition. Each ear is the ear of a back edge; if the back edge depends on an already oriented tree edge, the ear is determined and oriented. If the tree edge is not yet oriented the back edge is added to the set of dependent edges and processed as soon as the tree edge is oriented. It follows from the same argument as in the proof of Theorem 1 that all tree edges in the biconnected component of $\{s,t\}$ will eventually be oriented. □

Figure 5 shows the intermediate steps of the algorithm when applied to the running example. Note that the *st*-ordering is different from that obtained in [7], because, e.g., $d \hookrightarrow g$ is traversed before lowpoint-edge $d \hookrightarrow s$ (otherwise the order of c and d would be reversed).

An efficient implementation of Alg. 1 uses a doubly-linked list L, and stores all edges depending on a tree edge (including the edge itself) in a cyclic list that can be realized with an edge array (since these lists are disjoint). Since only the orientation of tree edges is needed, it is sufficient to store the orientation of the incoming DFS edge at each vertex. In total we need a doubly-linked vertex list, four vertex arrays (incoming edge, orientation of incoming edge, current child vertex, pointer to list position), two vertex stacks (DFS and ear traversal), and one edge array (next dependent edge). The edge array also serves to indicate whether an edge has already been traversed.

5 Discussion

We have presented a simple algorithm to compute an *st*-ordering of the vertices of a biconnected graph. It requires only a single traversal of the graph and maintains, after each step of the DFS, a maximum partial solution for the *st*-ordering, the *st*-orientation, and the open ear decomposition problem on the traversed subgraph.

Without modification, the algorithm can also be used to test for biconnectedness (simply check whether the length of the returned list equals the number of vertices in the graph), and it is readily seen from the inductive proof of Lemma 6 that resulting *st*-orderings have the following interesting property.

Corollary 3. *Algorithm 1 yields st-orderings in which the vertices of every subtree of the DFS tree form an interval.*

It is interesting to note that storing (tree) edge orientations at vertices corresponds to Tarjan's use of +/- labels in [7]. Since they can be interpreted as storing orientations at the parent rather than the child, they need to be updated, though. While the use of lowpoints eliminates the need to keep track of dependencies, lowpoints are known only after traversing the entire graph. Although Tarjan's algorithm [7] remains the most parsimonious, we thus feel that the eager algorithm is more flexible and intuitive.

Finally, the algorithm can be used to determine a generalization of bipolar orientations to non-biconnected graphs, namely acyclic orientations with a number of sources and sinks that is at most one larger than the number of biconnected components, and which is bipolar in each component. Whenever DFS backtracks over a tree edge that has not been orientated, it has completed a biconnected component. We are hence free to orient this tree edge any way we want and thus to determine, by recursively orienting dependent ears, a bipolar orientation of its entire component. The combined orientations are acyclic, and for each additional biconnected component we add at most one source or one sink, depending on how we choose to orient the first edge of that component and whether both its incident vertices are cut vertices.

Acknowledgments

I would like to thank Roberto Tamassia for initiating this research by making me aware of Gabow's work on path-based DFS, and Roberto Tamassia and Luca Vismara for very helpful discussions on the subject. Thanks to three anonymous reviewers for their detailed comments.

References

[1] J. Ebert. *st*-ordering the vertices of biconnected graphs. *Computing*, 30(1):19–33, 1983. 247, 248

[2] S. Even and R. E. Tarjan. Computing an *st*-numbering. *Theoretical Computer Science*, 2(3):339–344, 1976. 247, 248

[3] S. Even and R. E. Tarjan. Corrigendum: Computing an *st*-numbering. *Theoretical Computer Science*, 4(1):123, 1977. 247

[4] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74:107–114, 2000. 250

[5] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Proceedings of the International Symposium on the Theory of Graphs (Rome, July 1966)*, pages 215–232. Gordon and Breach, 1967. 248

[6] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1973. 249, 250

[7] R. E. Tarjan. Two streamlined depth-first search algorithms. *Fundamenta Informaticae*, 9:85–94, 1986. 247, 248, 249, 255, 256

[8] H. Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1932. 248