st-Orientations

September 29, 2005

1 Introduction

Let $G = (V, E)$ be an undirected biconnected graph of n nodes and m edges. The main problem this chapter deals with is different algorithms for orienting the edges of G. In fact, there are 2^m ways to achieve this. However, it is very useful in many applications to be able to produce *st*-oriented directed graphs which satisfy two distinct properties:

- They have one single source s and one single sink t
- They contain no cycles

Such an orientation of G's edges is called an *st*-orientation or a *bipolar* orientation (see figure 1). *St*-oriented graphs have many interesting properties. First of all, we can run

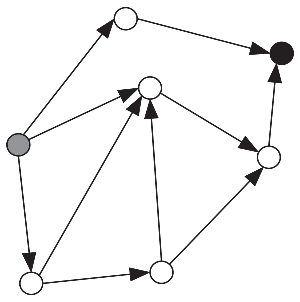

Figure 1: *st*-Orientation of a graph G.

several polynomial time algorithms on them (for example *longest path* and *topological sorting*) that we cannot run on undirected graphs and draw some useful conclusions. But how can we compute an *st*-orientation?Do all undirected graphs admit such an orientation?

2 *St*-Orienting the Edges

In 1967, Lempel, Even and Cederbaum [1] made a first approach to this problem, by presenting an algorithm for the computation of a numbering of the vertices of an undirected graph in order to check whether a graph is planar or not. They proved that, given any edge (s, t) of a biconnected graph G, the vertices of G can be numbered from 1 to n, so that vertex s receives number 1, vertex t receives number n and all other vertices are adjacent both to a lower-numbered and to a higher-numbered vertex. Actually, an undirected graph $G = (V, E)$ can be *st*-numbered if and only if the graph $G' = (V, E \cup (s, t))$ is biconnected. This numbering is called an *st*-numbering of G g. A formal definition of the *st*-numbering follows.

Let $G = (V, E)$ be an undirected biconnected graph. Let $\{s, t\}$ be one of each edges. An *st*-numbering is a function $g : V \to \{1, \ldots, n\}$ such that $g(s) = 1$, $g(t) = n$ and $\forall v \in V - \{s, t\}$ there are two edges (x, v) and (v, y) such that

$$g(x) < g(v) < g(y) \tag{2.1}$$

It is easy to prove that G has an *st*-orientation if and only if it has an *st*-numbering and we can compute either from the other in $O(m+n)$ time, as follows. Given an *st*-orientation, we number the vertices of G in topological order using Knuth's algorithm [2]. This produces an *st*-numbering. Given an *st*-numbering, we orient each edge from its lower-numbered to its higher-numbered endpoint. This produces an *st*-orientation.

2.1 The Tarzan-Even Algorithm

2.1.1 Preliminaries

In 1974, Even and Tarzan [3] developed an $O(m+n)$ algorithm for the computation of an st-numbering. The algorithm is based on depth first traversal and uses the circles formed during the execution of a DFS. As it is already known, given an undirected connected graph $G = (V, E)$ we can execute a DFS and get a DFS tree. All nodes v of the initial graph are contained in the tree and get a number $f(v)$ which actually denotes the rank of their visit.

A Depth First Search traversal separates the edges of our initial graph into two sets, the *tree* edges set U_t, with $|U_t| = n - 1$ and the *cycle* edges set U_c, with $|U_c| = m - n + 1$. U_t contains the edges that belong to the tree and U_c contains the remaining edges of the graph. Each edge $e \in U_c$ forms a circle. This edge always returns form a node x to a node y previously visited and forms a *basic* cycle. The collection C of all basic cycles is called a *basis* for the desired set of cycles (a basis set for a vector space is an appropriate analogy). A cycle edge (u, v) will be denoted with $u - ... - v$, whereas a tree edge (u, v), with $f(u) > f(v)$, (i.e. u is a child of v) will be denoted with $v \to u$. If a node v can be reached by u by following the tree path from node u to the root of the tree, we say that v is an ancestor of u and is denoted with $v \hookrightarrow u$. Note that for every cycle edge (u, v) of the DFS tree the following equivalence holds:

$$u - ... - v \Leftrightarrow u \hookrightarrow v \mid v \hookrightarrow u \tag{2.2}$$

As we said before, DFS forms a spanning tree, assigning a unique number $f(v)$ to every node v of the initial graph. These numbers are very crucial to the computation of an st-numbering as they define another function $\ell : V \to 1, \dots, n$. This function is called the *lowpoint* function and $\forall x \in V$ is defined as follows:

$$\ell(x) = \min(\{f(x)\} \cup \{f(y) : \exists w : x \hookrightarrow w \land w - ... - y\}) \tag{2.3}$$

Note that the lowpoint function is not an 1-1 function, i.e. there can be two nodes getting the same lowpoint. It is easy to see from 2.3 that a node x either gets its DFS number as a lowpoint or the DFS number of a node y, previously visited by DFS, reachable from x by following a downward tree path to a node w, which ends with a cycle edge from w to y. This path may contain no tree edges. Next, we will present a lemma that comes out of the definition of the lowpoint function.

Lemma 2.1. *If G is biconnected and $v \to w$, then $f(v) \neq 1$ implies $\ell(w) < f(v)$ and $f(v) = 1$ implies $\ell(w) = f(v) = 1$ [4].*

Proof. For the first case, when $f(v) \neq 1$, let c be a node above v in the DFS tree, i.e. $f(c) < f(v)$. As the graph is biconnected there must be a path from w to c not containing v. This path will certainly end with a back edge to c. Thus, c can be reached by w with a back edge and therefore it is $\ell(w) = f(c)$ and as $f(c) < f(v)$ it is $\ell(w) < f(v)$. For the second case, there is no other node with DFS number less than 1. Thus if $f(v) = 1$ and $v \to w$ then it must be $\ell(w) = f(v) = 1$. $\square$

The values $\ell(v)$ can easily be computed in time $O(m+n)$ during the execution of DFS. We will now describe the algorithm for the computation of an st-numbering. We are given an undirected biconnected graph $G = (V, E)$ and we want to assign numbers to its vertices which satisfy the definition of st-numbering. Let (s, t) be one edge of G. In the beginning,

Algorithm 1 Pathfinder(v)

```
1: if $\exists v - ... - w \in U_c$ new with $w \hookrightarrow v$ then
2:     mark $(v, w)$ as old;
3:     $p = \{v, w\}$;
4: else if $\exists v \rightarrow w \in U_t$ new then
5:     mark $(v, w)$ as old;
6:     $p = \{v, w\}$;
7:     while $w$ new do
8:         find new $(w, x)$ with $(f(x) = \ell(w) \mid (\ell(x) = \ell(w) \wedge w \rightarrow x))$;
9:         mark $w$ and $(w, x)$ as old;
10:        $p = p \cup (w, x)$;
11:        $w = x$;
12:    end while
13: else if $\exists v - ... - w \in U_c$ new with $v \hookrightarrow w$ then
14:    mark $(v, w)$ as old;
15:    $p = \{v, w\}$;
16:    while $w$ new do
17:        find new $(w, x)$ with $x \rightarrow w$;
18:        mark $w$ and $(w, x)$ as old;
19:        $p = p \cup (w, x)$;
20:        $w = x$;
21:    end while
22: else
23:    $p = \{\emptyset\}$;
24: end if
25: return $p$;
```

we execute a DFS, such that the root of the DFS tree is node t and the first edge of the tree is $t \rightarrow s$. During DFS, we also compute the lowpoint numbers $\ell(v)$ for every node v. The information generated by DFS is valuable for the remaining part of the algorithm.

The most important part of the algorithm is a procedure that, given a node v, returns a simple path from node v to a distinct node w. Initially, all nodes and edges of the graph are marked *new*, except nodes s, t and edge (s, t) that are marked *old*. Each successive call of the procedure *path(v)* returns a simple path of new edges and marks all vertices and edges contained in the path as *old*. Above we present the pseudocode of the algorithm (Algorithm1).

The procedure *path*(v) either produces a simple path of edges, which originates from node v to another node w, or returns the null path. When *path*(v) is called and the null path is returned, there are no other new edges emanating from node v, and thus the last part of the *if* statement is executed.

As referred above, *path*(v) is a procedure that is called by the main body of the algorithm. The main algorithm uses a stack, where the old vertices are stored. Initially the stack contains s on top of t. The top vertex on the stack, say v, is *deleted* and then *path*(v) is called. If *path*(v) returns a path $p = \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}\}$, then $v_{k-1}, v_{k-2}, \ldots, v_2, v_1$ are added to the top of the stack, where $v_1 = v$. Note that the last vertex of the path v_k is not added to the stack. If the null path is returned, then v is assigned the next available number and *not* put back on the stack. The pathfinder procedure is a simpler version of the one presented in [5].

Lemma 2.2. *Supppose 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. [3] □

2.1.2 The Algorithm

The main algorithm for the computation of an *st*-numbering uses the pathfinder procedure to compute an *st*-numbering. The pseudocode of the algorithm can be seen below: In the

Algorithm 2 Stnumber(G, s, t)

```
1: compute the lowpoints $\ell(v)$ for all nodes $v$;
2: mark $s$, $t$ and $(s, t)$ as old and all other vertices and edges as new;
3: initialize a stack $R$;
4: $R$ =push($t$);
5: $R$ =push($s$);
6: $i = 0$;
7: while $R \neq \emptyset$ do
8:    $v$=pop($R$)
9:    $p = \{\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{k-1}, v_k\}\}$ = Pathfinder($v$);
10:   if $p \neq \emptyset$ then
11:      for $j = k - 1$ downto 1 do
12:         $R$ =push($v_j$)
13:      end for
14:   else
15:      $i = i + 1$;
16:      $g(v) = i$;
17:   end if
18: end while
```

following, we will prove the correctness of our algorithm. The importance and efficiency of the algorithm depends on the clever use of the stack.

Theorem 2.3. *Algorithm* Stnumber *correctly computes an st-numbering of an undirected biconnected graph $G = (V, E)$.*

Proof. It is evident that no vertex v 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. Additionally, a vertex x finally receives a number when $path(x)$ returns the null path, i.e. all edges (x, w) for some w have been marked old.

Firstly, it is evident that vertex s receives number 1. This happens because s will always be on top of the stack until no new edges of type (s, w) exists. This time, $path(s)$ will return the null path and s will be the first vertex to permanently disappear from stack, thus receiving number one. The power of the stack lies in the fact that adjacent vertices in stack are adjacent vertices in the graph as well. Thus, an adjacent vertex of s, say r, will remain on top of stack until all edges emanating from r become old. No vertex will receive a number until r does. Thus r receives the next number. Vertex t finally receives number n. The procedure goes on and guarantees that every vertex $y \neq s, t$ will have at least one lower numbered adjacent vertex and at least one higher numbered adjacent vertex. $\square$

The running time of the *st* - numbering algorithm is $O(m+n)$ for the depth first search traversal plus the time required for the main body of the algorithm. The time required by the main body of the algorithm is dominated by the time spent in $Pathfinder()$ calls. The algorithm $Pathfinder()$ can be implemented so that a call requires time proportional to the number of edges found in the path. This requires that for each vertex v the following items are kept: a list of cycle edges $v - ... - w$ such that $v \hookrightarrow w$; a list of cycle edges $v - ... - w$ such that $w \hookrightarrow v$; a list of v's children; v's father; and finally an edge $\{v, w\}$ such that $f(w) = \ell(v) \mid \ell(w) = \ell(v)$. All these structures can be constructed during DFS and their storage requires linear space. Thus *path*() requires time $O(m+n)$, as each edge occurs in exactly one path, and therefore *st* - numbering takes time $O(m+n)$.
Let as now regard an undirected graph G of 10 vertices with adjacency matrix A:

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Table 1: The algorithm execution

iteration #	**stack status**	**path**	**operation**
1	{8,6}	$6 \to 1 \to 2 \to 4 \to 3 \to 9 \to 5 \to 7 \to 8$	
2	{8,7,5,9,3,4,2,1,6}	$6 \to 2$	
3	{8,7,5,9,3,4,2,1,6}	$6 \to 3$	
4	{8,7,5,9,3,4,2,1,6}	$6 \to 9$	
5	{8,7,5,9,3,4,2,1,6}	**null**	$g(6) = 1$
6	{8,7,5,9,3,4,2,1}	$1 \to 4$	
7	{8,7,5,9,3,4,2,1}	$1 \to 7$	
8	{8,7,5,9,3,4,2,1}	$1 \to 10 \to 9$	
9	{8,7,5,9,3,4,2,10,1}	**null**	$g(1) = 2$
10	{8,7,5,9,3,4,2,10}	$10 \to 2$	
...	...	...	...
21	{8,7,5}	**null**	$g(5) = 8$
22	{8,7}	**null**	$g(7) = 9$
23	{8}	**null**	$g(8) = 10$

Graph G is biconnected and thus we can apply our algorithm to find an st - numbering. We will find a $6-8$ - numbering. The reader can verify that during the algorithm execution the variables of table 1 will be computed. The final vector produced by the algorithm is

$$g_{68} = \left[\begin{array}{cccccccccc} 2 & 4 & 6 & 5 & 8 & 1 & 9 & 10 & 7 & 3 \end{array} \right]$$

Note that $g(6) = 1$ and $g(8) = 10$. Additionally, vector g satisfies the st-numbering definition.

2.2 A Streamlined Depth-First Search Algorithm

2.2.1 The Algorithm

Another simpler algorithm for the computation of an st-numbering was proposed by Tarzan in 1986 [6]. The algorithm is also based on a depth-first search traversal of the initial biconnected graph. In the depth first tree, we denote with $p(v)$ the father of node v. The algorithm works as follows.

It consists of two passes. The first pass is a depth first search during which for each vertex $v \in V$, $f(v), \ell(v)$ and $p(v)$ are computed. The second pass constructs a list L of the vertices, such that if vertices are numbered in the order they occur in L, an st-numbering results. Actually, 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 *mimus sign* (i.e., $s(u) = *-$), if u precedes v in L. Respectively, each vertex u that is a proper ancestor of the current vertex v has *plus sign* (i.e., $s(u) = *+$), if u follows v in L.

Initially $L = [s, t]$ and $s(s) = *-$. The second pass of the algorithm consists of repeating the following step for each vertex $v \neq s, t$ in preorder:

```
1: if $s(\ell(v)) == *+$ then
2:     Insert $v$ after $p(v)$ in $L$;
3:     $s(p(v)) = *-$;
4: end if
5: if $s(\ell(v)) == *-$ then
6:     Insert $v$ before $p(v)$ in $L$;
7:     $s(p(v)) = *+$;
8: end if
```

Theorem 2.4. *The st-numbering is correct.*

Proof. Consider the second pass of the algorithm. We must show that

- the signs assigned to the vertices have the claimed meaning
- if vertices are numbered in the order they occur in L, an st-numbering results.

For the first case, suppose $s = x_0$, $t = x_1, x_2, \ldots, x_l$ 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 $0 \leq i < j < l$, $s(x_i) = *+$ if and only if x_i follows x_j in L, i.e., $x_i = p(x_j)$. Since $s(x_k)$ 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. Hence the induction holds.

For the second case, let $v \neq s, t$. If $(v, \ell(v))$ is a back edge, the insertion of v between $p(v)$ and $\ell(v)$ 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 $\ell(w) = \ell(v)$. By lemma 2.1 we have that $\ell(v)$ is a proper ancestor of v, which means that $s(\ell(v))$ remains constant during the time v and w are added to L. It follows that v appears between $p(v)$ and w in the completed list L, which implied that in the numbering corresponding to L, v is adjacent to both a lower-numbered and higher-numbered vertex. Thus, the second case holds. □

It is obvious that the algorithm runs in linear time $O(m + n)$.

2.2.2 An Example

Following, we give an execution example of the algorithm. Suppose we want to compute

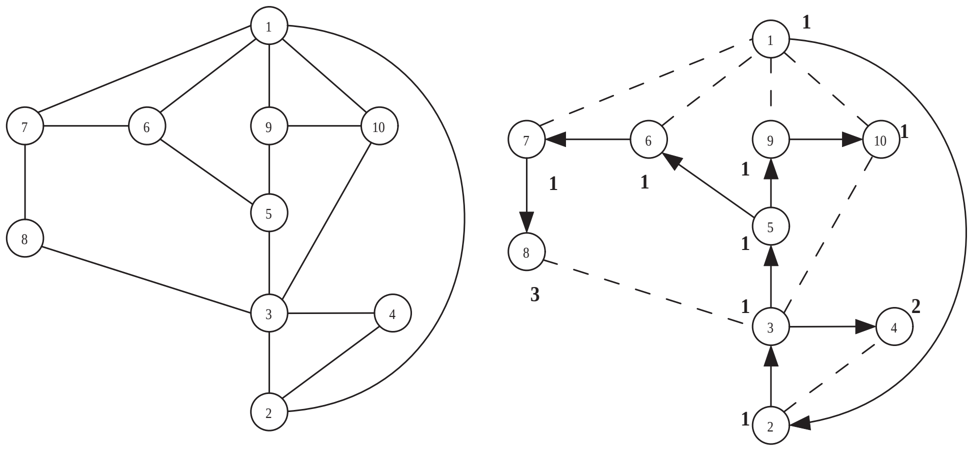

Figure 2: A biconnected graph G and its depth first search tree.

a 2-1 numbering of the biconnected graph of figure 1. First we execute a DFS, and we compute the DFS tree and the lowpoint values. The lowpoint values of each vertex in Figure 2 are depicted with bold numbers.

Table 2: The algorithm execution. Irrelevant signs are omitted

iteration #	**vertex added v**	**List L**
1	-	$\{1^-, 2\}$
2	3	$\{1^-, 3, 2^+\}$
3	4	$\{1^-, 3^-, 4, 2^+\}$
4	5	$\{1^-, 5, 3^+, 4, 2^+\}$
5	6	$\{1^-, 6, 5^+, 3^+, 4, 2^+\}$
6	7	$\{1^-, 7, 6^+, 5^+, 3^+, 4, 2^+\}$
7	8	$\{1^-, 7^-, 8, 6^+, 5^+, 3^+, 4, 2^+\}$
8	9	$\{1^-, 7, 8, 6, 9, 5^+, 3^+, 4, 2^+\}$
9	10	$\{1^-, 7, 8, 10, 6^+, 9^+, 5^+, 3^+, 4, 2^+\}$

2.3 An Algorithm for Direct st-Orientation Computation

2.3.1 Preliminaries

In the previous sections, two algorithms were presented which compute an st-oriented directed graph by using an st-numbering. In this section, we present an algorithm that directly computes an st-orientation. Additionally, the algorithm simultaneously computes an st-numbering and orienting the edges from lower-numbered vertices to higher numbered vertices. Hence, no topological sorting to the computed st-oriented directed graph should be executed in order to extract an st-numbering. In order to describe the algorithm we give some necessary definitions.

Let $G = (V, E)$ be a one-connected graph. Suppose now that G consists of a set of blocks B and a set of cutpoints C. Set B contains blocks whereas set C contains nodes of the original graph. The respective block-cutpoint tree $T = (B \cup C, U)$ has $|B| + |C|$ nodes and $|B| + |C| - 1$ edges. Additionally, its structure is defined as follows

$$(i, j) \in U \Leftrightarrow i \in B \wedge j \in C \wedge j \in V(i) \vee j \in B \wedge i \in C \wedge i \in V(j) \tag{2.4}$$

where $V(k)$ denotes the vertex set of a block k. The above relation shows that the edges of the block-cutpoint tree always connect pairs of blocks and cutpoints in a way that in every edge the participating cutpoint exists in the vertex set of the participating block.

The above definition implies that the block-cutpoint tree is a free tree, i.e., it has no distinct root. In order to define the *rooted* block-cutpoint tree we must define a vertex t to be the root. In that case if t is a cutpoint, the block-cutpoint tree is rooted on t else if t is not a cutpoint the block-cutpoint tree is rooted on the block that contains t (see figure 3).

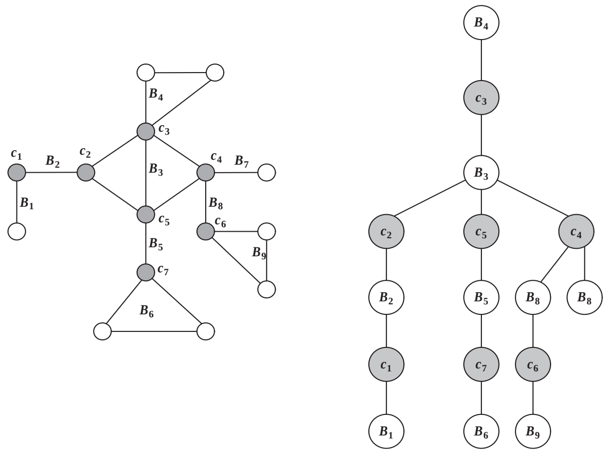

Figure 3: A one-connected graph and its block-cutpoint tree rooted on B_4.

Finally, we define the leaf-blocks of the block-cutpoint tree to be the blocks except for the root of the block-cutpoint tree that are defined by a sole cutpoint, i.e., they contain a sole cutpoint. The block-cutpoint tree can be computed in time $O(m+n)$ with a recursive algorithm similar to DFS [5].

Lemma 2.5. *Let $G = (V, E)$ be an n-node undirected biconnected graph and s, t be two of its nodes. Suppose we remove s and all its incident edges. Then there is at least one* *neighbor of s lying in each leaf-block the block-cutpoint tree. Moreover, this neighbor is not a cutpoint.*

Proof. If the graph $G - \{s\}$ is still biconnencted, the proof is trivial, as the block-cutpoint tree is made up from one sole node (the biconnected component $G - \{s\}$), which is both the root and the leaf-block of the tree.

If the graph $G - \{s\}$ is one-connected, suppose that there is a leaf-block ℓ of the block-cutpoint tree defined by the cutpoint c such that $N(s) \cap \ell = \emptyset$. Then c, if removed, still disconnects G and thus G is not biconnected, which does not hold. The same occurs if $N(s) \cap \ell = \{c\}$. Hence there is always at least one neighbor of s lying in each leaf-block of the block-cutpoint tree, which is not a cutpoint. $\square$

Corollary 2.6. *Let $G = (V, E)$ be an n-node undirected biconnected graph and s, t be two of its nodes. Suppose we remove s and all its incident edges. Then there are at least two neighbors of s lying at leaf-blocks of the block-cutpoint tree. Moreover, these neighbors are not cutpoints.*

Proof. As it happens with every free tree, the block-cutpoint tree has at least two leaf-blocks. By lemma 2.5, each leaf will contain at least one neighbor of s that is not a cutpoint. Hence, in total, there will always be at least two neighbors of s lying in some leaf-blocks of the block-cutpoint tree. $\square$

2.3.2 The Algorithm

Lemma 2.5 is the basis for the development of the algorithm. Suppose we are given a biconnected graph $G = (V, E)$ and we want to compute an *st*-orientation of it. The idea here is to root the block-cutpoint tree either on t or on the biconnected component that contains t and to repeatedly deal with the leaf-blocks (biconnected components) of the tree until we reach sink t. In this way we give a certain direction to the edges of the graph.

Suppose we repeatedly produce the graphs $G_{i+1} = G_i - \{v_i\}$ for all $i = 1, \dots, n$ and $G_1 = G$. Initially, $v_1 = s$. During the procedure we always maintain a block-cutpoint tree rooted on node t. Additionally, we maintain a structure Q that plays a major role in the choice of the current source. Q initially contains the desired source for the final orientation, s. Finally we maintain the leaf-blocks for the block-cutpoint tree rooted on t. Node v_i is chosen such that

$$v_i \in B_i^\ell \cap Q \sim h_i^\ell, \; i = 1, \dots, n-1 \tag{2.5}$$

where B_i^ℓ is the ℓ-st leaf-block of the rooted block-cutpoint tree at iteration i and h_i^ℓ is the cutpoint that defines B_i^ℓ. Note that in the case that $i = 1$ the biconnected component is the initial biconnected graph and the cutpoint that defines it is the desired sink of the orientation, t. When a source v_i is removed from the graph, we have to update Q in order to be able to choose our next source. Q is then updated as follows

$$Q = Q \cup \{N_{G_i}(v_i) \sim t\} - \{v_i\}, \; i = 1, \dots, n-1 \tag{2.6}$$

where $N_G(x)$ denotes the neighborhood of node x at graph G. This procedure is continued until Q gets empty. Suppose now $F = (V, E')$ is a directed graph derived by the described procedure, which we call **STN**, by setting

$$E' = \bigcup_{i=1}^{n-1} ((v_i, N_{G_i}(v_i)) \tag{2.7}$$

We claim that F is an st-oriented directed graph.

Lemma 2.7. *During STN, every node becomes a source exactly once. Additionally, after exactly $n-1$ iterations (i.e. after all nodes but t have been processed), it is $Q = \emptyset$ (i.e. Q gets empty).*

Proof. Let $v \neq t$ be a node that never becomes a source. This means that all incident edges (u, v) have direction $u \rightarrow v$. As the algorithm gradually removes sources, by simultaneously assigning direction, one u must be a cutpoint (as $v \neq t$ will become a biconnected component of a sole node). But all nodes u are chosen to be neighbors of prior sources. By corollary 2.6, u can never be a cutpoint, which does not hold. Hence node $v \neq t$ will certainly become a source exactly once, as it immediately removed after becoming a source. In the beginning of iteration $n-1$, the remaining graph will be consisted of only two nodes, namely v_{n-1} and t, as at each step of STN a node is removed from the original graph, which is simultaneously removed from Q. Hence, in the beginning of iteration $n-1$ it will be $Q = \{v_{n-1}\}$. Q is then updated as follows

$$Q = Q \cup \{N_{G_{n-1}}(v_{n-1}) \sim t\} - \{v_{n-1}\}$$

and as $N_{G_{n-1}}(v_{n-1}) \sim t = \emptyset$ the result follows. To complete the proof we must show that during all iterations $i = 1, \ldots, n-2$ it is $Q \neq \emptyset$. This is so because neighbors of a node that belongs to a biconnected component, which are least two, are inserted into Q and hence Q will always contain at least one element, as t is never inserted into Q. $\square$

Corollary 2.8. *Let $B_1^k, B_2^k, \ldots, B_r^k$ be the leaf-blocks of the block-cutpoint tree that is produced after the removal of vertex v_{k-1}. If the leaf-blocks are defined by the cutpoints $h_1^k, h_2^k, \ldots, h_r^k$ then $\forall i = 1, \ldots r$ it is $|B_i^k \cap Q \sim h_i^k| \geq 1$.*

Proof. This corollary ensures that there will always exist a choice to continue the execution of STN. The proof follows immediately from Lemma 2.5, Corollary 2.6 and Lemma 2.7. $\square$

Lemma 2.9. *The directed graph $F = (V, E')$ has exactly one source s and exactly one sink t.*

Proof. Node $v_1 = s$ is indeed a source, as all edges $(v_1, N(v_1))$ are assigned the same direction (from v_1 to its neighbors) at the first step. Note t is indeed a sink as it is never being chosen to become a current source and all its incident edges are assigned a direction from its neighbors to it during prior iterations of STN. We have to prove that all other nodes have at least one incoming and one outcoming edge. As all nodes $v \neq t$ become sources exactly once, there will be at least one u such that $(v, u) \in E'$. Sources $v \neq t$ are actually nodes that have been inserted into Q during a prior iteration of the algorithm. Before being chosen to become sources, all nodes $v \neq s \neq t$ are put into Q as neighbors of prior sources and thus there is at least one u such that $(u, v) \in E'$. Hence F has exactly one source and exactly one sink. $\square$

Lemma 2.10. *The directed graph $F = (V, E')$ has no cycles.*

Proof. Suppose after STN has ended and F has been derived, there is a directed cycle consisted by the nodes $v_1, v_2, \ldots, v_k, v_1$. This means that $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1) \in E'$. During STN, after an edge (v_j, v_{j+1}) is inserted into E', v_j is deleted from the graph and never processed again (Lemma 2.7) and v_{j+1} is inserted into Q so that it becomes a future source. In our case after edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ will have been given direction, nodes $v_1, v_2, \ldots, v_{k-1}$ will have been deleted from the graph. To create a cycle, v_1 should be inserted into Q as a neighbor of v_k, which does not hold as $v_1 \notin N_{G_k}(v_k)$ (v_1 has already been deleted from the graph). Thus F has no cycles. □

Theorem 2.11. *The directed graph $F = (V, E')$ is st-oriented.*

Proof. Immediate from the the two previous lemmas. □

Following, we give the algorithm pseudocode. Note that the algorithm returns both the *st*-oriented graph F and the *st*-numbering vector g.

Algorithm 3 STN(G, s, t)

```
Q = {s};
i = 0;
Initialize F = (V', E') to be the final directed graph;
Initialize the block-cutpoint tree T to be graph G; Its cutpoint is sink t;
while $Q \neq \emptyset$ do
  for all leaf-blocks $B_\ell^{h_\ell}$ do
    $i = i + 1$;
    find $v_\ell \in B_\ell \cap Q \sim \{h_\ell\}$ {next source}
    $g(v_\ell) = i$; {g is the st-numbering vector}
    $V = V - \{v_\ell\}$ {a source is removed from G}
    $V' = V' \cup \{v_\ell\}$ {and is added to F}
    for all edges $(v_\ell, i) \in E$ do
      $E = E - \{(v_\ell, i)\}$
      $E' = E' \cup \{(v_\ell, i)\}$
    end for
    $Q = Q \cup \{N(v_\ell) \sim t\} - \{v_\ell\}$ {the set of possible sources}
  end for
  $T(t, B_1^{h_1}, B_2^{h_2}, \ldots, B_r^{h_r})$=UpdateBlocks(G);
end while
return F, g;
```

Theorem 2.12. *Algorithm STN runs in $O(mn)$ time.*

Proof. During the algorithm execution there are $n - 1$ node removals. Additionally, the time of each iteration is wholly consumed by the procedure that updates the block-cutpoint tree, which is $O(m+n)$. Hence the algorithm runs in $(n-1)O(m+n) = O(mn)$ time. □

However, there are cheaper methods to update the block-cutpoint tree. If we use techniques proposed in [7], we can drop this bound to $O(m \log^5 n)$ time:

Theorem 2.13. *By using the algorithm for the biconnectivity maintenance proposed in [7], algorithm STN can be implemented to run in $O(m \log^5 n)$ time.*

Proof. During the algorithm execution there all m edges of the graph are removed. Additionally, the time of each iteration is wholly consumed by the procedure that updates the block-cutpoint tree, which in [7] is $O(\log^5 n)$ (per edge deletion). Hence the algorithm runs in $mO(\log^5 n) = O(m \log^5 n)$ time. □

2.3.3 An Example

Following we present an example of the algorithm execution:

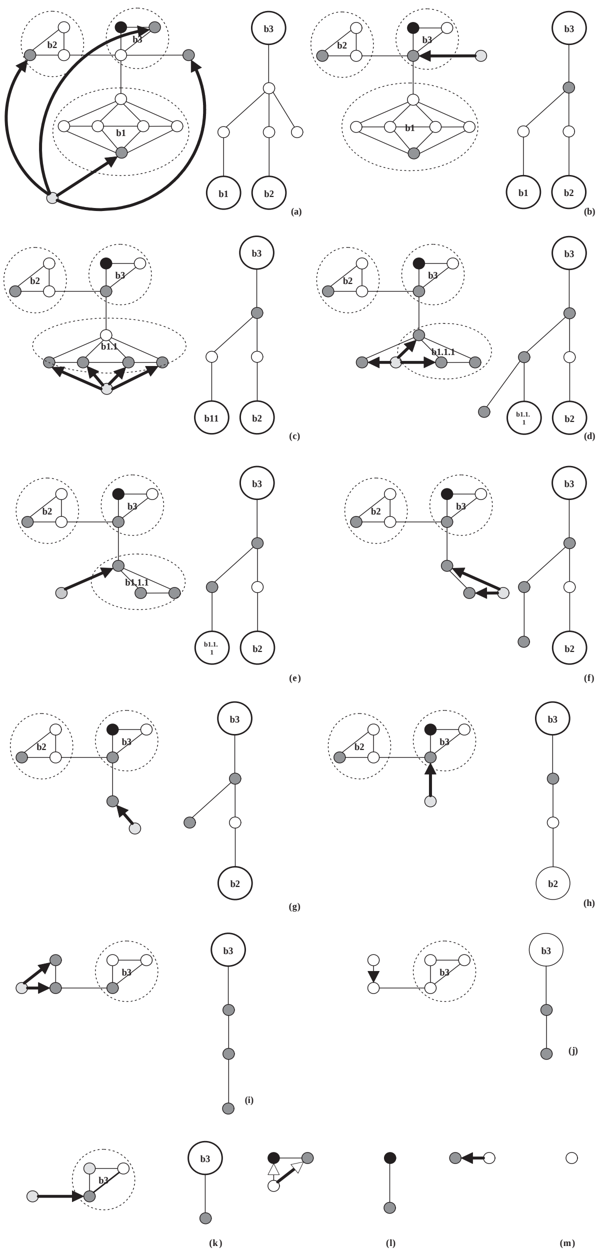

Figure 4: The execution of the algorithm.

The algorithm presented allows us to influence the value of some parameters that characterize the final *st*-oriented graph. For example the choice we make every time on Q is very crucial for the longest path value of the final *st*-oriented graph.

References

[1] A. Lempel, S. Even, 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

[2] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*,Third Edition, Addison-Wesley, Reading, MA 1997

[3] S. Even, R. Tarzan, *Computing an st-numbering* Theoretical Computer Science 2 (1976) 339-344

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

[5] J. Hopcroft, R. Tarzan, *Efficient algorithms for Graph Manipulation*, Comm. ACM 16(1973) 372-378

[6] R. Tarzan, *Two Streamlined Depth-First Search Algorithms*, Fundamenta Informatica IX (1986) 85-94 North-Holland

[7] J. Holm, K. De Lichtenberg, M. Thorup, *Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity*