Theoretical Computer Science 408 (2008) 224–240

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

# Algorithms for computing a parameterized *st*-orientation[☆]

Charalampos Papamanthou[a,*], Ioannis G. Tollis[b,c]

[a] Department of Computer Science, Brown University, Providence RI, USA

[b] Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece

[c] Department of Computer Science, University of Crete, Heraklion, Greece

ARTICLE INFO

*Keywords:*
Graph algorithms
Planar graphs
*st*-numberings
Longest path

ABSTRACT

*st*-orientations (*st*-numberings) or bipolar orientations of undirected graphs are central to many graph algorithms and applications. Several algorithms have been proposed in the past to compute an *st*-orientation of a biconnected graph. In this paper, we present new algorithms that compute such orientations with certain (parameterized) characteristics in the final *st*-oriented graph, such as the length of the longest path. This work has many applications, including Graph Drawing and Network Routing, where the length of the longest path is vital in deciding certain features of the final solution. This work applies to other difficult problems as well, such as graph coloring and of course longest path. We present extended theoretical and experimental results which show that our technique is efficient and performs well in practice.



## 1. Introduction

The problem of orienting an undirected graph such that it has one source, one sink, and no cycles (*st*-orientation) is central to many graph algorithms and applications, such as graph drawing [2–6], network routing [7,8] and graph partitioning [9]. Most implemented algorithms use any algorithm that produces such an orientation, e.g., [10], without expecting any specific properties of the oriented graph. In this paper we present new techniques that produce such orientations with specific properties. Namely, our techniques are able to control the length of the longest path of the resulting directed acyclic graph. This provides significant flexibility to many graph algorithms and applications [2,3,7–9,1,11].

Given a biconnected undirected graph $G = (V, E)$, with $n$ vertices and $m$ edges, and two nodes $s$ and $t$, an *st*-numbering [10] of $G$ is a numbering of its vertices such that $s$ receives number 1, $t$ receives number $n$ and every other node except for $s, t$ is adjacent to at least one lower-numbered and at least one higher-numbered node. An *st*-orientation of $G$ is defined as an orientation of its edges, such that a directed acyclic graph with exactly one source $s$ and exactly one sink $t$ is produced. There is a direct relation between *st*-numberings and *st*-orientations: an *st*-orientation of an undirected graph can be easily computed using an *st*-numbering of the respective graph $G$ and orienting the edges of $G$ from *low* to *high*.

*st*-numberings were first introduced in 1967 in [12], where it is proved that given any edge $\{s, t\}$ of a biconnected undirected graph $G$, we can define an *st*-numbering. The proof of a theorem in [12] gives a recursive algorithm that runs in time $O(nm)$. However, in 1976 Even and Tarjan proposed an algorithm that computes an *st*-numbering of an undirected biconnected graph in $O(n + m)$ time [10]. Ebert [13] presented a slightly simpler algorithm for the computation of such a numbering, which was further simplified by Tarjan [14]. The planar case has been extensively investigated in [15], where

☆ This work was supported in part by INFOBIOMED code: IST-2002-507585 and the Greek General Secretariat for Research and Technology under Program "ARISTEIA", Code 1308/B1/3.3.1/317/12.04.2002. An early version of this paper appeared in the 14th Symposium on Graph Drawing (GD2006) [C. Papamanthou, I.G. Tollis, Parameterized st-orientations of graphs: Algorithms and experiments, in: Proc. Int. Conf. Graph Drawing, 2006, pp. 220–233].

* Corresponding author.
*E-mail address:* cpap@cs.brown.edu (C. Papamanthou).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.08.012

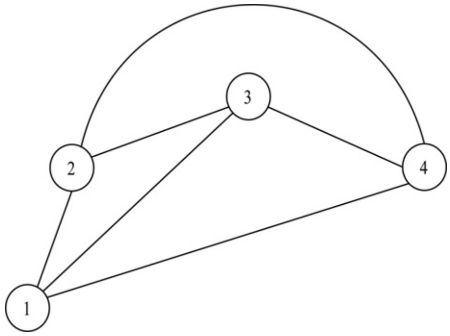



**Fig. 1.** A 1–4 recursively biconnected graph on $P = 1, 2, 3, 4$. Note that if edge $(2, 4)$ was to be removed, the graph would not be recursively biconnected any more.

a linear time algorithm is presented which may reach any $st$-orientation of a planar graph. Additionally, in [16] a parallel algorithm is described (running in $O(\log n)$ time using $O(m)$ processors) and finally in [17], another linear time algorithm for the problem is presented. An overview of bipolar orientations is presented in [18].

Developing yet another algorithm for simply computing an $st$-orientation of a biconnected graph would probably seem meaningless, as there already exist many efficient linear time algorithms for the problem [10,13,14]. In this paper we present a new algorithm, along with theoretical and experimental results that show that there is an efficient way to control the length of the longest path that corresponds to an $st$-numbering. The importance of this research direction has been implied in the past [3,15]. Our algorithms are able to compute $st$-oriented graphs of "user-defined" length of longest path, the value of which is very important in the quality of the solution many algorithms produce. For example the area-bounds of many graph drawing algorithms [2,3,6] are utterly dependent on the length of the longest path of the $st$-oriented graph. Additionally, network routing via $st$-numberings gives alternate paths towards any destination, and therefore deriving different (parameterized longest-path) $st$-numberings provides flexibility to many proposed routing protocols [7,8].

The paper is organized as follows: in Section 2 we present the new algorithm and give a formal proof of its correctness. In Section 3 we discuss longest path parameterized $st$-orientations. Section 4 presents the experimental results for various classes of graphs and finally Section 5 gives conclusions, open problems and future research directions.

## 2. A new algorithm for computing an *st*-orientation

### 2.1. General

In this section, we present an algorithm that computes an $st$-orientation of a biconnected graph $G = (V, E)$. We analyze its behavior and give a proof of correctness. This algorithm is designed in such a way that makes it possible to control the length of the longest path of the final $st$-oriented graph. For the rest of the paper, $n = |V|$, $m = |E|$, $N_G(v)$ denotes the set of neighbors of node $v$ in graph $G$, $s$ is the source of the graph, $t$ is the sink of the graph and $l(u)$ is the length of the longest path of a node $u$ from the source $s$ of the graph. We begin by presenting the algorithm's behavior on a special class of graphs and then we present its extension to general graphs.

### 2.2. A special case

In this section, we describe an algorithm for computing an $st$-orientation of a special class of graphs. This class includes graphs that *maintain* their biconnectivity after successive removals of vertices (for example the $K_n$ graphs).

**Definition 1.** Let $G = (V, E)$ be an undirected biconnected graph. We say that $G$ is $st$-recursively biconnected on $P$ if there is a permutation of vertices $P = v_1, v_2, \ldots, v_n$ with $v_1 = s$ and $v_n = t$ such that the graphs $G_i = G_{i-1} - \{v_{i-1}\}$, $v_i \in N_{G_{i-1}}(v_{i-1}) - \{t\}$, $i = 2, \ldots, n-1$ and $G_1 = G$ are biconnected.

An example of a recursively biconnected graph is depicted in Fig. 1. We now present a lemma that gives an algorithm for the transformation of an $st$-recursively biconnected undirected graph to an $st$-oriented graph.

**Lemma 2.** *Let $G = (V, E)$ be an undirected $st$-recursively biconnected graph on $P = v_1, v_2, \ldots, v_n$ with $v_1 = s$ and $v_n = t$. Then the set of directed edges*

$$E' = \{(v_1, N_{G_1}(v_1)), (v_2, N_{G_2}(v_2)), \ldots, (v_{n-1}, N_{G_{n-1}}(v_{n-1}))\}$$

*forms an $st$-oriented graph.*

**Proof.** We prove the lemma by giving an algorithm for $st$-orienting an $st$-recursively biconnected graph. Suppose we remove one by one the nodes on $P$ starting with $v_1 = s$. Each time we remove a node, it becomes a current source of the remainder of the graph, and all its incident edges are oriented away from it. First we must prove that, beginning with $v_1$, we can reach every node $v_i$, $i \geq 2$. Suppose there is a node $v_k$ that is never reached by a previously removed node. This can be done only if the removal of adjacent nodes disconnects a graph $G_l$, $l < k$. This is not true, as all graphs $G_i$ are biconnected, and hence all nodes will finally be removed from the graph by following neighbors of previously removed nodes. It remains to see that the directed graph produced by following this procedure is *st*-oriented. First of all, the oriented graph has a single source $v_1$ (all its incident edges are oriented away from it) and a single sink $v_n$ (all its incident edges have been oriented in prior iterations towards it). Also, there is no other source and sink due to the definition of the recursive biconnectivity. Finally, suppose there existed a cycle. Then, after the removal of a vertex $v_i$, we would have to process a vertex $v_k$ with $k < i$ (this is the only way for a cycle to be formed). But all vertices $v_k$ with $k < i$ have already been removed. Hence there is no cycle and the graph is *st*-oriented. □

### 2.3. General graphs case

In the previous section, we examined a special class of graphs. However, most graphs are not recursively biconnected and even if they are, it is generally hard to find such a permutation $P$. We now present the general case, where there is no other option than removing a node that produces a one-connected subgraph. Before continuing with this section, we will introduce some useful intuition and terminology.

#### 2.3.1. The very first approach

The aim of this work is the computation of *st*-orientations of longest path length that can be efficiently controlled by an input parameter. Towards this goal, we investigated the possibility of modifying the existing linear algorithms, in order to produce longest path parameterized *st*-orientations. These algorithms, such as [10], proceed by choosing one vertex at a time. This means that at each iteration, they maintain and update a set of vertices (that is computed according to some biconnectivity criteria), and continue their execution by processing neighbors of a chosen vertex. Thus in order to produce multiple *st*-orientations using these algorithms, one should consider different combinations of vertex sequences. Some heuristics applied on the Even–Tarjan algorithm are described in [19], where after extensive computational results, we reached the conclusion that using different vertex sequences, produces *st*-orientations that have almost **no** difference in the longest path length. After similar attempts on other existing algorithms, it became evident that linear time was not enough to produce both a correct *st*-orientation and to be able to discriminate between different longest path length *st*-orientations.

#### 2.3.2. Exploiting biconnectivity

The main idea behind the algorithm is the exploitation of the biconnectivity structure of a graph. Let $G = (V, E)$ be a one-connected undirected graph, i.e., a graph that contains at least one vertex whose removal causes the initial graph to disconnect. The vertices that have that property are called *separation vertices*, *articulation points* or *cutpoints*. Each one-connected graph is composed of a set of blocks (biconnected components) and cutpoints that form a tree structure. This tree is called the *block-cutpoint tree* of the graph and its nodes are the blocks and cutpoints of the graph. Suppose now that $G$ consists of a set of blocks $B$ and a set of cutpoints $C$. The respective block-cutpoint tree $T = (B \cup C, U)$ has $|B| + |C|$ nodes and $|B| + |C| - 1$ edges. The edges $(i, j) \in U$ of the block-cutpoint tree always connect pairs of blocks and cutpoints such that the cutpoint of a tree edge belongs to the vertex set of the corresponding block (see Fig. 2).

The block-cutpoint tree is a free tree, i.e., it has no distinct root. In order to transform this free tree into a rooted tree, we define the $t$-rooted block-cutpoint tree with respect to a vertex $t$. Consequently, the root of the block-cutpoint tree is the block that contains $t$ (see Fig. 2). Finally, we define the leaf-blocks of the $t$-rooted block-cutpoint tree to be the blocks, except for the root, of the block-cutpoint tree that contain a single cutpoint. The block-cutpoint tree can be computed in $O(n + m)$ time with an algorithm similar to DFS [20]. Next, we give some results that are necessary for the development of the algorithm.

**Lemma 3.** *Let $G = (V, E)$ be an undirected biconnected graph and $s$, $t$ be two of its nodes. Then there is at least one neighbor of $s$ lying in each leaf-block of the $t$-rooted block-cutpoint tree of $G - \{s\}$. Moreover, this neighbor is not a cutpoint.*

**Proof.** If graph $G - \{s\}$ is still biconnected, the proof is trivial, as the $t$-rooted block-cutpoint tree consists of a single node (the biconnected component $G - \{s\}$), which is both root and leaf-block of the $t$-rooted block-cutpoint tree. If graph $G - \{s\}$ is one-connected (see Fig. 3), suppose that there is a leaf-block $\ell$ of the $t$-rooted block-cutpoint tree defined by cutpoint $c$ such that $N(s) \cap \ell = \{\emptyset\}$. Then $c$, if removed, still disconnects $G$ and thus $G$ is not biconnected, a contradiction. 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 $t$-rooted block-cutpoint tree, which is not a cutpoint. □

The main idea of the algorithm is based on the successive removal of nodes and the simultaneous update of the $t$-rooted block-cutpoint tree. We call each such node a *source*, because at the time of its removal, it is effectively chosen to be a source of the remainder of the graph. We initially remove $s$, the first source, which is the source of the desired *st*-orientation and give direction to all its incident edges from $s$ to all its neighbors. After this removal, there exist three possibilities:

- The graph remains biconnected
- The graph is decomposed into several biconnected components but the number of leaf-blocks remains the same
- The graph is decomposed into several biconnected components and the number of leaf-blocks changes.

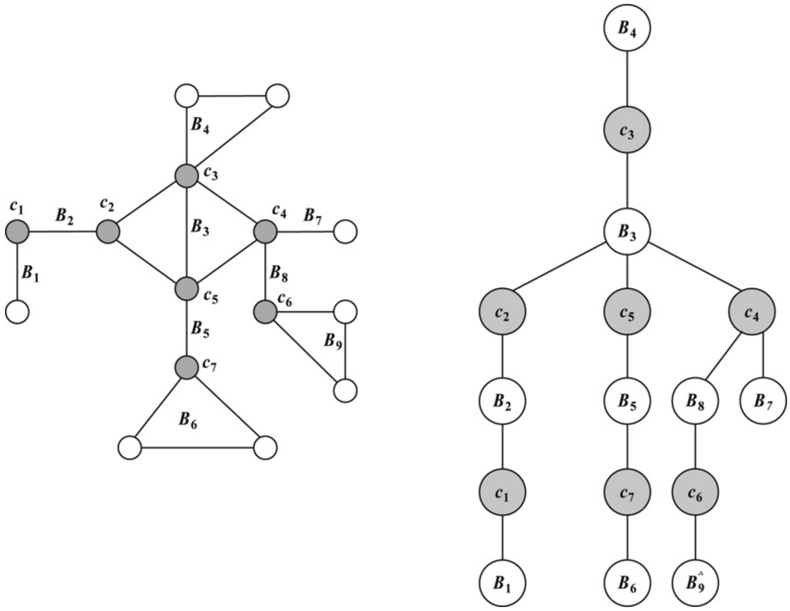



**Fig. 2.** A one-connected graph and the $t$-rooted block-cutpoint tree rooted at $B_4$.

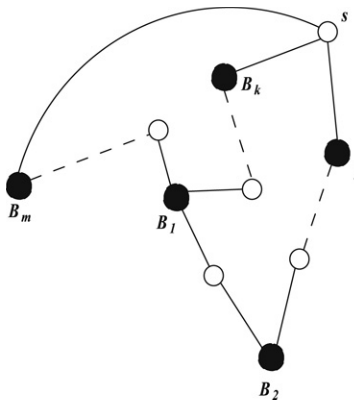



**Fig. 3.** Proof of Lemma 3.

This procedure continues until all nodes of the graph but one are removed. Finally, we encounter the desired sink, $t$, of the final $st$-orientation. The updated biconnectivity structure gives us information about the choice of our next source. Actually, the biconnectivity maintenance allows us to remove nodes and simultaneously maintain a "map" of possible vertices whose future removal may or may not cause dramatic changes to the structure of the tree.

As it will be clarified in the next sections, at every step of the algorithm there will be a set of potential sources to continue the execution. Our aim is to establish a connection between the current source choice and the length of the longest path of the produced $st$-oriented graph.

### 2.3.3. *The algorithm*

Now we describe the algorithm in a more formal way. We name this procedure STN. Let $G = (V, E)$ be an undirected biconnected graph and $s, t$ two of its nodes. We will compute an $st$-orientation of $G$. Suppose we recursively produce the graphs $G_{i+1} = G_i - \{v_i\}$, where $v_1 = s$ and $G_1 = G$ for all $i = 1, \ldots, n-1$.

During the algorithm we always maintain a $t$-rooted block-cutpoint tree. 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, $s$. For the generalized version of the algorithm, where we only want to ensure correctness of the final $st$-orientation, $Q$ can be implemented as simple *set* data structure. As we will see later, for the sake of a *parameterized st*-orientation, $Q$ should be implemented as a priority queue. Finally we maintain the leaf-blocks of the $t$-rooted block-cutpoint tree. During the $i$th iteration of the algorithm node $v_i$ is chosen so that

- it is not a cutpoint node that belongs to $Q$
- it belongs to a leaf-block of the $t$-rooted block-cutpoint tree.

Note that for $i = 1$ there is a single leaf-block (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 by removing $v_i$ and by inserting all of the neighbors of $v_i$ except for $t$.

Each time a node $v_i$ is removed we orient all its incident edges from $v_i$ to its neighbors. The procedure continues until $Q$ becomes empty. Let $F = (V', E')$ be the directed graph computed by this procedure. We claim that $F = (V', E')$ is an $st$-oriented graph.

**Lemma 4.** *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), $Q$ becomes empty.*

**Proof.** Let $v \neq t$ be a node that never becomes a source. This means that all incident edges $(u, v)$ have direction $u \to v$. 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 single node). But all nodes $u$ are chosen to be neighbors of prior sources. By Lemma 3, $u$ can never be a cutpoint, hence node $v \neq t$ will certainly become a source exactly once. Finally, $Q$ becomes empty at the end of the algorithm, as the algorithm chooses $n$ times from $Q$ and nodes stored in $Q$ are distinct. □

By combining Lemmas 3 and 4, we see that at each iteration of the algorithm there will be at least one node to be chosen as a future source:

**Corollary 5.** *Suppose after vertex $v_{k-1}$ is removed, $r$ different leaf-blocks are created. Then in each leaf-block of the $t$-rooted block-cutpoint tree there exists at least one non-cutpoint node that belongs to $Q$.*

**Lemma 6.** *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 a direction from $v_1$ to its neighbors in the first step. Node $t$ is indeed a sink as it is never 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 outgoing edge. As all nodes $v \neq t$ become sources exactly once, there will be at least one node $w$ such that $(v, w) \in E'$. 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 inserted into $Q$ as neighbors of prior sources and thus there is at least one $w$ such that $(w, v) \in E'$. Hence $F$ has exactly one source and one sink. □

**Lemma 7.** *The directed graph $F = (V', E')$ has no cycles.*

**Proof.** Suppose STN has ended and there is a directed cycle $v_j, v_{j+1}, \ldots, v_{j+l}, v_j$ in $F$. This means that $(v_j, v_{j+1}), (v_{j+1}, v_{j+2}), \ldots, (v_{j+l}, v_j) \in E'$. During STN, after an edge $(v_k, v_{k+1})$ is inserted into $E'$, $v_k$ is deleted from the graph and never processed again and $v_{k+1}$ is inserted into $Q$ so that it becomes a future source. In our case after edges $(v_j, v_{j+1}), (v_{j+1}, v_{j+2}), \ldots, (v_{j+l-1}, v_{j+l})$ will have been oriented, nodes $v_j, v_{j+1}, \ldots, v_{j+l-1}$ will have been deleted from the graph. To create a cycle, $v_j$ should be inserted into $Q$ as a neighbor of $v_{j+l}$, which does not hold as $v_j \notin N_{G_{j+l}}(v_{j+l})$ ($v_j$ has already been deleted from the graph). Thus $F$ has no cycles. □

By Lemmas 6 and 7 we have:

**Theorem 8.** *The directed graph $F = (V', E')$ is $st$-oriented.*

In Algorithm 1 we present STN in pseudocode. During the execution of the algorithm, we can also compute an $st$-numbering $f$ (line 9) of the initial graph. Actually, for each node $v_i$ that is removed from the graph, the subscript $i$ is the final $st$-number of node $v_i$. The $st$-numbering can however be easily computed in linear time after the algorithm has ended, by executing a topological sorting on the computed $st$-oriented graph $F$.

Note that in the algorithm we use a vector $m(v)$ (line 17), where we store a timestamp for each node $v$ of the graph that is inserted into $Q$. These timestamps will be of great importance during the choice of the next candidate source, and will give us the opportunity to control the length of the longest path. Actually, they express the last time that a node $v$ becomes candidate for removal.

Regarding the time complexity of the algorithm, the recursion is executed exactly $n - 1$ times, and the running time of each recursive call is consumed by the procedure that updates the block-cutpoint tree, which is $O(n + m)$ [20]. Hence it is easy to conclude that STN runs in $O(nm)$ time. However, it can be made to run faster by a more efficient algorithm to maintain biconnectivity.

In fact, Holm, Lichtenberg and Thorup [21] investigated the problem of maintaining a biconnectivity structure without computing the block-cutpoint tree from scratch. They presented a fully dynamic algorithm that supports the insertion and deletion of edges and maintains biconnectivity in $O(\log^5 n)$ amortized time per edge insertion or deletion. In our case, only deletions of edges are done. If we use this algorithm in order to keep information about biconnectivity, STN can be implemented to run in $O(m \log^5 n)$. Moreover, for planar graphs, we can compute biconnected components in $O(\log n)$ amortized time per edge deletion due to [22]. Hence, the algorithm can be implemented to run in $O(m \log n)$ time for planar graphs. Hence, we obtain the following:

**Theorem 9.** *Given a biconnected graph $G = (V, E)$ of $n$ vertices and $m$ edges, a source $s \in V$ and a sink $t \in V$,* STN$(G, s, t)$ *can be implemented to run in $O(m \log^5 n)$ time. Moreover, if the graph is planar,* STN$(G, s, t)$ *can be implemented to run in $O(m \log n)$ time.*

**Algorithm 1** STN$(G, s, t)$

Initialize $F = (V', E')$;
Initialize $m(i) = 0$ for all nodes $i$ of the graph; (timestamp vector)
$j = 0$; {*Initialize a counter*}
$Q = \{s\}$; {*Insert s into Q*}
**STREC**$(G, s)$; {*Call the recursive algorithm*}
————————————————————————
**function STREC**$(G, v)$
$j = j + 1$;
$f(v) = j$;
$V = V - \{v\}$; {*A source is removed from G*}
$V' = V' \cup \{v\}$; {*and is added to F*}
**for all** edges $(v, i) \in E$ **do**
  $E = E - \{(v, i)\}$;
  $E' = E' \cup \{(v, i)\}$;
**end for**
$Q = Q \cup \{N(v) \sim \{t\}\} - \{v\}$; {*The set of possible next sources*}
$m(N(v)) = j$;
**if** $Q == \{\emptyset\}$ **then**
  $f(t) = n$;
  **return**;
**else**
  $T(t, B_j^1, B_j^2, \ldots, B_j^r) =$ **UpdateBlocks**$(G)$; {*Update the t-rooted block-cutpoint tree; $h_j^i$ is the cutpoint that defines the leaf-block $B_j^i$*}
  **for all** leaf-blocks $(B_j^i, h_j^i)$ **do**
    ***choose*** $v_\ell \in B_j^\ell \cap Q \sim \{h_j^\ell\}$
    **STREC**$(G, v_\ell)$;
  **end for**
**end if**

The $st$-orientation algorithm defines an $st$-*tree* $T_s$. Its root is the source of our graph $s$ $(p(s) = -1)$. It can be computed during the execution of the algorithm. When a node $v$ is removed, we simply set $p(u) = v$ for every neighbor $u$ of $v$, where $p(u)$ is a pointer to the father of each node $u$. Note that the father of a vertex can be updated many times until the algorithm terminates. This tree is a directed tree that has two kinds of edges, the tree edges, which show the *last* father–ancestor assignment between two nodes made by the algorithm and the non-tree edges that include all the remaining edges. The non-tree edges never produce cycles. Finally, note that the sink $t$ is always a leaf of the $st$-tree $T_s$.

**Algorithm 2** STN$(G, s, t)$

$Q = \{s\}$; {*insert s into Q*}
$j = 0$;{*Initialize a counter*}
Initialize $F = (V', E')$;
Initialize the $t$-rooted block-cutpoint tree $T$ to be graph $G$; Its cutpoint is sink $t$;
**while** $Q \neq \emptyset$ **do**
  **for** all leaf-blocks $B_j^i$ **do**
    $j = j + 1$;
    ***choose*** $v_\ell \in B_j^\ell \cap Q \sim \{h_j^\ell\}$ {*$h_j^\ell$ is the cutpoint that defines $B_j^\ell$*}
    $f(v_\ell) = j$;
    $V = V - \{v_\ell\}$ {*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**
  $T(t, B_j^1, B_j^2, \ldots, B_j^r) =$ **UpdateBlocks**$(G)$;
**end while**
**return** $F, g$;

As it happens with every $st$-oriented graph, there is a directed path from every node $v$ to $t$ and hence the maximum depth of the $st$-tree will be a lower bound for the length of the longest path, $l(t)$:

**Theorem 10.** *Let $G$ be an undirected biconnected graph and $s$, $t$ two of its nodes. Suppose we run* STN *on it and we produce the $st$-oriented graph $F$ and its $st$-tree $T_s$. If $d(T_s)$ denotes the maximum depth of the $st$-tree then $l(t) \geq d(T_s)$.*

In Fig. 4, the algorithm execution on a biconnected graph $G$ is depicted. In Fig. 5, we can see the final $st$-oriented graph $F$ and the respective $st$ tree $T_s$. Algorithm 1 can also be implemented non-recursively. Actually, for large-size graphs, we can only use the non-recursive algorithm (Algorithm 2) in order to avoid stack overflow problems.

Algorithm 2 works as follows. It does not update the $t$-rooted block-cutpoint tree at every iteration (see line 18). After the first node is removed, it updates the $t$-rooted block-cutpoint tree and it removes one node from each leaf-block. That means that it actually calls the biconnectivity update procedure, only after all the leaf-blocks have been processed.

Finally, we must make an important remark. Instead of each time processing nodes that belong to the leaf-blocks of the $t$-rooted block-cutpoint tree, we could process non-cutpoint nodes that belong to some block of the $t$-rooted block-cutpoint tree. It is easy to prove that there will always exist such a node, and therefore all the Lemmas presented before would certainly apply to this case as well. However, choosing nodes that belong to the leaf-blocks of the $t$-rooted block-cutpoint tree gives us the opportunity to control the length of the longest path of the final directed graph more efficiently.

## 3. Longest path parameterized *st*-orientations

### 3.1. General

As stated in the previous section, our algorithm aims at producing $st$-oriented graphs of predefined longest path length. There are exponentially many $st$-oriented graphs that can be produced for a certain biconnected undirected graph, and it is desirable to be able to influence the length of the longest path by taking advantage of the *freedom* of choice the algorithm gives us.

Observe that the key in determining the length of the final longest path is the sequence of sources the algorithm uses. These sources are non-cutpoint nodes that belong both to $Q$ and to a leaf-block of the $t$-rooted block-cutpoint tree.

Hence during iteration $j$ of the algorithm, we have to pick a leaf-block of the $t$-rooted block-cutpoint tree (say the $l$-st) and we always have to make a choice on the structure (see line 8 of the Algorithm 2):

$$Q' = B_j^l \cap Q \sim \{h_j^l\}.$$

We have used two approaches in order to produce $st$-oriented graphs with long longest path length, and $st$-oriented graphs with small longest path length. During each iteration of the algorithm, a timer $j$ (line 7 of Algorithm 2) is incremented and each vertex $x$ that is inserted into $Q$ gets a timestamp $m(x) = j$.

Our investigation has revealed that if vertices with high timestamp are chosen, then long sequences of vertices are formed and thus there is high likelihood to obtain a long longest path. We call this way of choosing vertices maxSTN. Actually, maxSTN resembles a DFS traversal (i.e., it searches the graph at a *maximal* depth). Hence, during maxSTN, the next source $v$ is arbitrarily chosen from the set

$$\{v \in Q' : m(v) = \max\{m(i) : i \in Q'\}\}.$$

On the contrary, we have observed that if vertices with low timestamp are chosen, then the final $st$-oriented graph has relatively small longest path. We call this way of choosing vertices minSTN, which in turn resembles a BFS traversal. Hence, during minSTN, the next source $v$ is arbitrarily chosen from the set

$$\{v \in Q' : m(v) = \min\{m(i) : i \in Q'\}\}.$$

Note that the above sets usually contain more than one element. This means that ties exist and have to be broken. Breaking the ties in both cases is very important in determining the length of the longest path. Finally, for efficiency reasons, we can implement $Q$ as a priority queue.

Additionally, the length of the longest path from the source $s$ of the final directed graph to the currently removed node $u$ is immediately determined (when $u$ is removed, i.e., $u$ enters the sink set $W$) and cannot change during future iterations of the algorithm. This happens because during $u$'s removal, the direction of all its incident edges is determined, and there is no way to reach $u$ with a path that includes nodes that have not yet been removed (and that would probably change $l(u)$). Hence, we can either execute the longest path algorithm on the so far produced $sW$-DAG (where $W$ is a set of sinks), or apply a relaxation method during the execution of the algorithm (see in next sections), and compute $l(u)$:

**Remark 11.** Suppose a node $u$ is removed from the graph during STN and at this time we run the longest path algorithm to the so far produced $sW$-DAG, getting a longest path length from $s$ to $u$ equal to $l(u)$. The longest path length from $s$ to $u$ in the final $st$-DAG is also $l(u)$.

This remark is important because it gives us an idea of how the developed algorithm can relate to the length of the longest path.

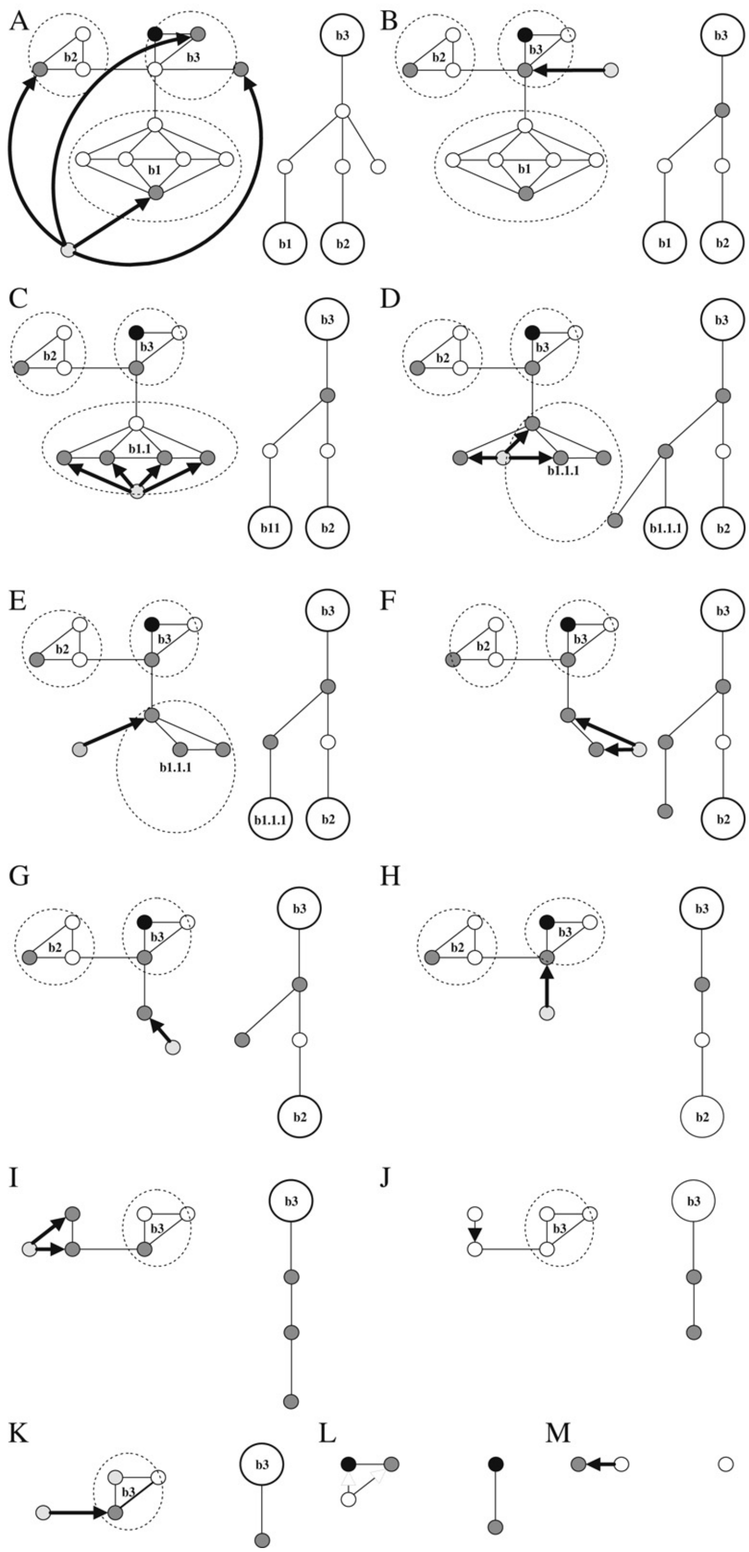



**Fig. 4.** The algorithm execution. At each iteration of the algorithm (A–M), the graph and the block-cutpoint tree are depicted.

In order to have an upper bound on the length of the longest path of a biconnected graph, we are going to present our longest path results for a special class of biconnected graphs that have an a priori length of longest path equal to $n - 1$:

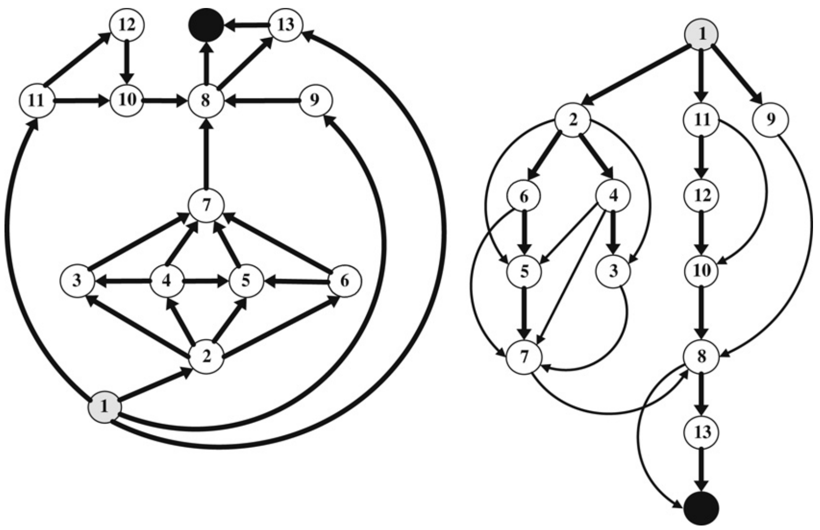



**Fig. 5.** The final $st$-oriented graph (left) and the $st$-tree $T_s$ (right).

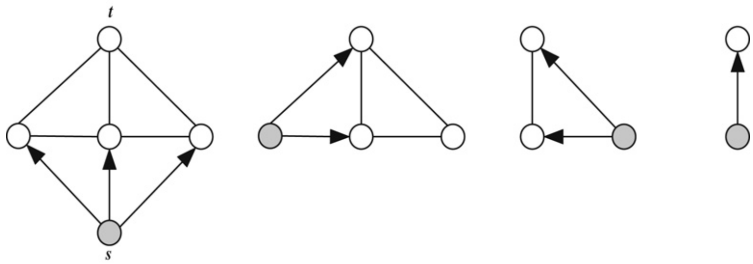



**Fig. 6.** Choosing vertices with minSTN for a biconnected component that remains biconnected throughout the execution of the algorithm. Note that at the third execution of the algorithm, the middle vertex is not chosen so that a small longest path length $st$-orientation can be achieved.

**Definition 12.** Let $G = (V, E)$ be an undirected biconnected graph and let $s, t$ be two distinct vertices of $G$. Graph $G$ is $st$-Hamiltonian if it admits a Hamiltonian path having $s$ and $t$ as its endvertices.

### 3.2. Maximum case (maxSTN)

**Lemma 13.** *Let $G = (V, E)$ be an undirected st-Hamiltonian graph.* maxSTN *computes an st-oriented graph with length of longest path equal to $n - 1$ if and only if the t-rooted block-cutpoint tree is a path (of blocks and cutpoints).*

**Proof.** For the direct, suppose maxSTN computes an $st$-oriented graph of maximum longest path length $n - 1$ and at some iteration of STN a vertex $v$ is removed and the block-cut point tree is decomposed into a tree that has more than one (say $k$) leaves. Then, there are $k$ different directed paths from vertex $v$ to the final sink $t$ of the graph. The longest path cannot be the union of these paths, because all these paths have orientations towards $t$. Hence $l(t) < n - 1$, contradiction. For the inverse, suppose that the produced length of longest path is less than $n - 1$. This means that at some iteration $i$ of the algorithm a source $v$ of timestamp $j < i$ is removed. In this case the source removed before $v$ must belong to a leaf-block other than the leaf-block of $v$, because if they belonged to the same leaf-block, $v$ would have a timestamp equal to $i$. By hypothesis, only a single leaf-block is maintained, which does not hold. Hence $l(t) = n - 1$. □

Note that the inverse holds only for the case of the maxSTN procedure. Fig. 6 provides a counter example, showing that if the general STN procedure is applied, a Hamiltonian path cannot always be achieved, even if a single leaf-block is maintained. Hence, we come to the conclusion that in order to produce an $st$-oriented graph with long longest path, one necessary condition is to maintain a single leaf-block of the $t$-rooted block-cutpoint tree. We will see later (in the Complexity Issues section) that this is an NP-hard problem.

maxSTN tries to mimic the DFS traversal of a graph, as it tries to explore the current biconnected component at a maximal depth. In this way, long paths of vertices are created, which are more likely to contribute to a longer longest path of the final directed graph, something that is illustrated in the experimental results section. If maxSTN could choose vertices in a way that the maximum sequence of vertices is created, then we could probably compute an $st$-oriented graph with maximum longest path. Instead, maxSTN "approximates" the long paths by creating different *individual* paths of vertices. An *individual* path of vertices $P_r$ computed by our algorithm is defined as follows: suppose the algorithm enters the $k$-th iteration and $k - 1$

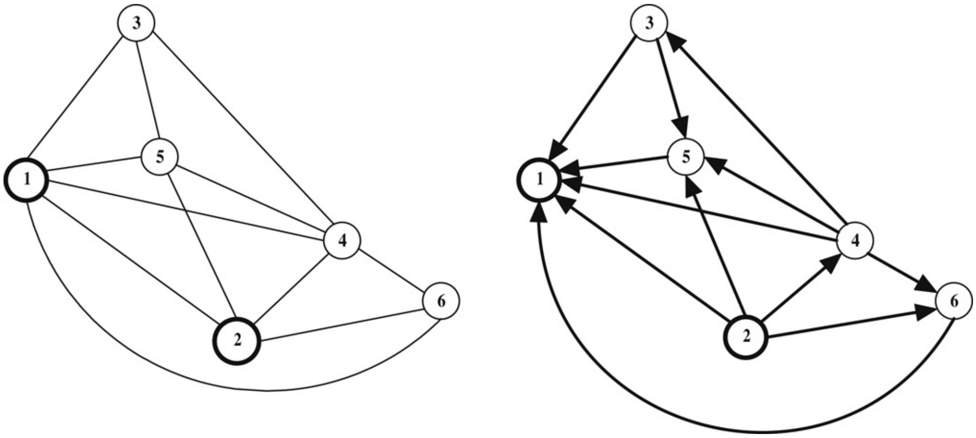


**Fig. 7.** maxSTN applied to a 2–1 Hamiltonian graph. No optimal DAG is produced (longest path length = 4).

vertices of the graph have been removed with the following order: $v_1, v_2, \dots, v_{k-1}$. All $r$ individual paths $P_1, P_2, P_3, \dots, P_r$ can be computed during the execution of the algorithm as follows. Initially we insert the first vertex removed into the first path ($v_1 >> P_1$). Suppose $v_j$ ($j < k$) is removed and $r$ different paths have been created till this iteration. Vertex $v_j$ has a timestamp $m(v_j)$. To decide if $v_j$ will be added to the current individual path $P_i$ or to a next (new) path $P_{i+1}$, we execute the following algorithm:

**if** $m(v_j) < m(v_{j-1}) + 1$ **then**
    $i = i + 1$;
**end if**
$v_j >> P_i$;

Actually, when the creation of a new path begins (i.e., when $m(v_j) < m(v_{j-1})+1$), we say that maxSTN *backtracks*. The length of the longest path of the final *st*-oriented graph is strongly dependent on the number of times that maxSTN backtracks. All these observations lead to the following remark:

**Remark 14.** Suppose STN enters iteration $j$. $m(v_j) < m(v_{j-1}) + 1$ implies that all nodes $v \in Q$ with $m(v) = j = \max\{m(i) : i \in Q\}$ do not belong to $Q'$.

The longest path length of the final directed graph will be the union of pieces of some of the created individual paths (hence $l(t) \geq \max_{i=1,\dots,r}\{|P_i|\}$), that achieves the largest number of successive (neighboring) vertices and can be computed in polynomial time during or after the algorithm execution (by applying some relaxation method).

Fig. 7 depicts the execution of the algorithm for a 6-node 2–1 Hamiltonian graph. The vertices are chosen by the algorithm in the following order: 2, 4, 3, 5, 6, 1. Note that two leaf-blocks are created and that's why the final longest path length is not optimal. If node 6 were chosen first, an *st*-oriented graph with maximum longest path length would be computed. During the execution of the algorithm, two paths are created, the path 2, 4, 3, 5, 1 and the path 6, 1. The final longest path is the first path.

### *3.3. Minimum case (*minSTN*)*

minSTN is a procedure that computes *st*-oriented graphs with relatively small length of the longest path. In this section, we give some theoretical results that justify this assumption.

minSTN works exactly the same way as maxSTN with the difference that it *backtracks* for a different reason. As we saw before, maxSTN creates long directed paths of vertices and it backtracks when it encounters a cutpoint (no matter if its timestamp is the maximum one), which is prohibited by the algorithm to be chosen as a next source. In maxSTN, the criterion of backtracking is: *If you encounter a cutpoint, continue execution from the node with the maximum timestamp*. On the other hand, minSTN works as follows: It creates small paths of vertices because backtracking occurs more often, as nodes of minimum timestamp usually lie on previously explored paths (see Fig. 8). Actually suppose during the execution of minSTN $r'$ such paths of vertices $P_1, P_2, P_3, \dots, P_{r'}$ are created. These paths can be computed with exactly the same algorithm that computes the maxSTN paths, with the difference that the case $m(v_j) < m(v_{j-1}) + 1$ is likely to occur more times during minSTN than during maxSTN.

### *3.4. Longest path computations*

Generally, the length of the longest path computed by the STN algorithm is also connected with the structure of the $t$-rooted block-cutpoint tree. Next, we investigate the connection between the length of the longest path of the resulting directed graph, and the number of leaf-blocks that are produced during the execution of the algorithm.

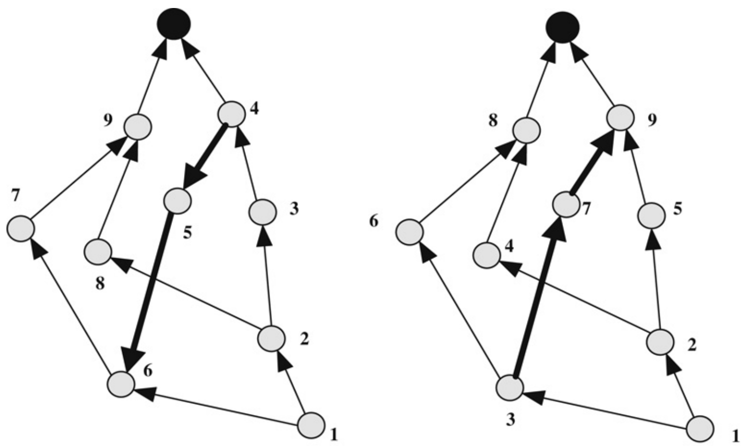


**Fig. 8.** maxSTN (left) and minSTN (right) applied to the same biconnected component. The black node is a cutpoint. The thick lines show the different orientation that results in different length of the longest path. The number besides the node represents the visit rank of each procedure.

**Theorem 15.** *Suppose* STN *is run on an undirected st-Hamiltonian graph G. Let $k_i$ denote the number of the leaf-blocks of the t-rooted block-cutpoint tree after the i-th removal of a node, for $i = 1, 2, \dots, n-1$. Then $l(t) \le n - 1 - \sum_{k_i > k_{i-1}} (k_i - k_{i-1})$.*

**Proof.** Suppose the $i$-th iteration of the algorithm begins. Then node $v_i$ is removed. The removal of $v_i$ gives a block-cutpoint tree of $k_i$ leaf-blocks. When an iteration $i$ causes the increase of the leaf-blocks from $k_{i-1}$ to $k_i$, then, in the best case, there are at least $k_i - k_{i-1}$ nodes that for sure will not participate in the final longest path. Hence we can derive an upper bound for $l(t)$ that equals the maximum longest path that can be achieved minus the number of vertices which are lost for sure, i.e., $l(t) \le n - 1 - \sum_{k_i > k_{i-1}} (k_i - k_{i-1})$. $\square$

In the experiments conducted on $st$-Hamiltonian graphs we have observed that the length of the longest path computed by maxSTN is usually very close to $n - 1 - \sum_{k_i > k_{i-1}} (k_i - k_{i-1})$.

Also, during STN, there are formed two sets of nodes $R, R'$ with $V = R \cup R'$. $R$ contains the nodes that have have been removed from the graph whereas $R'$ contains the nodes that have not yet been removed. All edges $(v, x)$ such that $v \in R$ have already been oriented and hence the directed paths leading to all nodes $v \in R$ have been determined. That is why the length of the longest path from $s$ to a removed node $v \in R$ is immediately determined at the time of its removal (Remark 14).

Actually, if we apply a relaxation algorithm during STN, we can compute the longest path length $l(v)$ from s to every node $v \in R$ during the execution of STN. This can be achieved as follows: in the beginning, we initialize the longest path vector $l$ to be the zero vector, hence

$$l(v) = 0 \quad \forall v \in V.$$

Suppose that at a random iteration of the algorithm we remove a node $u \in R'$, and we orient all $u$'s incident edges $(u, i)$ away from $u$. For every oriented edge $(u, i) \in E'$ we relax $l(i)$ as follows:

**for all** $(u, i) \in E'$ **do**
  **if** $l(i) < l(u) + 1$ **then**
    $l(i) = l(u) + 1$;
  **end if**
**end for**

This relaxation is exactly the same used by the algorithm that computes *longest paths* in directed acyclic graphs. Note that nodes $i$ belong to $R'$, and hence all nodes that belong to $Q$ (or $Q'$) will have an updated value $l(i)$ different than zero. Additionally, at the time a node $v$ is removed from the graph (and enters $R$), its longest path length $l(v)$ is always equal to $l(v') + 1$, where $v'$ is a node that had previously removed from the graph. Suppose now we enter the $k$-th iteration of the algorithm and $v_k$ is removed. Let

$$M_k = \max\{l(v_j) : j = 1, \dots, k\}$$

where $M_k$ denotes the maximum longest path length computed by STN till iteration $k$. All the observations presented lead to the following Lemma:

**Lemma 16.** *Suppose* STN *enters iteration k and $v_k$ is removed. Then $M_j \le M_{j-1} + 1$ for all $j = 2, \dots, k$.*

Actually, Lemma 16 points out the fact that when STN enters iteration $k$, no dramatic changes can happen to the maximum longest path length computed till iteration $k$. The increase is always at most one unit. This is actually happening when $v_k$ has a previously removed neighbor $v_l$, $l < k$ and $(v_l, v_k) \in E'$, such that $l(v_l) = M_{k-1}$. If there is no such node, it holds $M_k = M_{k-1}$ and no increase is observed.

### 3.5. Longest path timestamps and weighted graphs

Until now, we have defined the timestamps in accordance with a current timer $j$, which is updated during the execution of the algorithm: Each node $v$ inserted into $Q$ is associated with a timestamp value $m(v)$, which is set equal to $i$, every time that $v$ is discovered by a removed node $v_i$, i.e., $v$ is a neighbor of $v_i$. We call this method *current timestamp* method.

There is however another way to define the timestamps. As we saw in the previous section, during the execution of the algorithm we can compute (using the relaxation method) the longest path length from $s$ to each processed node $u$. We call this method the *longest path timestamp* method and it works as follows. Each node $v$ inserted into $Q$ is associated with a timestamp value $m(v)$, which is set equal to the *relaxed* longest path length $l'(v)$, which is lower than the final longest path length $l(v)$ (this is determined by the time of $v$'s removal). As we will discuss later, it has been experimentally observed, that the current timestamp method is a more efficient way to control the length of the longest path of the final directed graph.

The longest path timestamp method can be used to produce long or short $st$-orientations of weighted graphs. The presented algorithm, implemented with the longest path timestamp method can be used to compute weighted numberings on the weighted $st$-oriented graph that is produced. Let $c_{uv}$ be the weights of the graph edges $(u, v) \in E$. Suppose we update the longest path lengths using the following algorithm:

**for all** $(u, i) \in E'$ **do**
  **if** $l(i) < l(u) + c_{iu}$ **then**
    $l(i) = l(u) + c_{iu}$;
  **end if**
**end for**

Then we can use the computed longest paths to update the timestamps and implement the algorithm for weighted graphs as well.

### 3.6. Computational complexity issues

In this section, we will investigate some issues concerning the complexity of the developed algorithm. First of all it is easy to see that maintaining a block-cutpoint tree of a sole leaf-block during STN is NP-hard.[1] The proof comes from the fact that if we could do so, we could apply maxSTN (see Lemma 13) to an $st$-Hamiltonian graph and find its longest path, which is a well known NP-hard problem [23]. Following we define two decision problems and prove their NP-hardness.

**Definition 17.** Given an undirected biconnected graph $G = (V, E)$, two of its nodes $s, t$, an integer bound $k$, can we transform $G$ to an $st$-oriented graph $F$ than contains a longest path of length at least k?

**Theorem 18.** *The Maximum st-Oriented Graph Problem is NP-hard.*

**Proof.** We reduce the $st$-Directed Hamilton Path, which is NP-complete [23], to it. The $st$-directed Hamilton Path problem seeks an answer to the following yes/no question: given a directed graph $G = (V, E)$ and two vertices $s, t$ is there a directed path from $s$ to $t$ that visits all vertices exactly once? The polynomial reduction follows. Given an instance $G' = (V', E'), s', t'$ of the $st$-directed Hamilton Path problem, count the number $|V'|$ of nodes of $G'$ and output the instance $G = G', k = |V'|$, $s = s', t = t'$ for the maximum longest path length $st$-oriented graph problem. Obviously, $G$ has a simple directed path of length $k = |V'|$ from $s$ to $t$ if and only if $G'$ has a directed Hamilton path from $s'$ to $t'$. □

**Definition 19.** Given an undirected biconnected graph $G = (V, E)$, two of its nodes $s, t$, an integer bound $k$, can we transform $G$ to an $st$-oriented graph $F$ than contains a longest path of length at most k?

This problem is also NP-hard, as shown in [24].

### 3.7. Inserting parameters into the algorithm

As it has already been reported, it would be desirable to be able to compute $st$-oriented graphs of length of longest path within the interval $[\lambda(t), \ell(t)]$. This is called a parameterized $st$-orientation. So the question that arises is: can we insert a parameter into our algorithm, for example a real constant $p \in [0, 1]$ so that our algorithm computes an $st$-oriented graph of length of longest path that is a function of $p$?

This is feasible if we modify STN. As the algorithm is executed exactly $n$ times ($n$ vertices are removed from the graph), we can execute the procedure maxSTN for the first $pn$ iterations and the procedure minSTN for the remaining $(1-p)n$ iterations. We call this method parSTN($p$) and we say that it produces an $st$-oriented graph with length of longest path from $s$ to $t$ equal to a function $\Delta(p)$. Note that parSTN(0) is equivalent to minSTN, thus $\Delta(0) = \lambda(t)$ while parSTN(1) is equivalent to maxSTN and $\Delta(1) = \ell(t)$. parSTN has been tested and it seems that when applied to $st$-Hamiltonian graphs (biconnected graphs that contain at least one path from $s$ to $t$ that contains all the nodes of the graph) there is a high likelihood that $\Delta(p) \geq p(n-1)$. Actually, $\Delta(p)$ is very close to $p(n-1)$. Additionally, it has been observed that if we switch the order of maxSTN and minSTN execution, i.e., execute minSTN for the first $pn$ iterations and maxSTN for the remaining $(1-p)n$ iterations, it is usually the case that $\Delta(p) \leq p(n-1)$. In this case, $\Delta(p)$ is again very close to $p(n-1)$.

[1] Actually, it is NP-hard to decide whether or not the removal of a vertex $v_i$ will cause a future decomposition of the block-cutpoint tree into more than one leaf-blocks.

**Table 1**
Results for parameterized *st*-orientations of density 3.5 *st*-Hamiltonian graphs

| $n$ | $p=0$ | | $p=0.3$ | | $p=0.5$ | | $p=0.7$ | | $p=1$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ |
| 100 | 14.00 | 0.141 | 38.90 | 0.393 | 59.20 | 0.598 | 76.50 | 0.773 | 92.20 | 0.931 |
| 200 | 18.60 | 0.093 | 74.10 | 0.372 | 113.00 | 0.568 | 147.90 | 0.743 | 186.60 | 0.938 |
| 300 | 23.30 | 0.078 | 104.80 | 0.351 | 165.10 | 0.552 | 219.20 | 0.733 | 280.70 | 0.939 |
| 400 | 23.30 | 0.058 | 139.10 | 0.349 | 213.80 | 0.536 | 289.30 | 0.725 | 376.30 | 0.943 |
| 500 | 29.20 | 0.059 | 169.40 | 0.339 | 267.30 | 0.536 | 361.20 | 0.724 | 470.70 | 0.943 |
| 600 | 27.90 | 0.047 | 202.10 | 0.337 | 318.90 | 0.532 | 428.90 | 0.716 | 566.60 | 0.946 |
| 700 | 30.90 | 0.044 | 231.60 | 0.331 | 369.40 | 0.528 | 499.00 | 0.714 | 663.40 | 0.949 |
| 800 | 30.00 | 0.038 | 264.90 | 0.332 | 415.30 | 0.520 | 566.50 | 0.709 | 755.60 | 0.946 |
| 900 | 31.70 | 0.035 | 294.30 | 0.327 | 469.90 | 0.523 | 640.20 | 0.712 | 848.10 | 0.943 |
| 1000 | 36.20 | 0.036 | 322.10 | 0.322 | 518.20 | 0.519 | 709.30 | 0.710 | 940.00 | 0.941 |
| 1100 | 38.90 | 0.035 | 353.90 | 0.322 | 576.30 | 0.524 | 782.90 | 0.712 | 1033.40 | 0.940 |
| 1200 | 34.40 | 0.029 | 387.00 | 0.323 | 622.10 | 0.519 | 845.50 | 0.705 | 1127.80 | 0.941 |
| 1300 | 34.30 | 0.026 | 421.10 | 0.324 | 674.50 | 0.519 | 917.00 | 0.706 | 1223.10 | 0.942 |
| 1400 | 38.90 | 0.028 | 448.80 | 0.321 | 718.40 | 0.514 | 983.90 | 0.703 | 1319.90 | 0.943 |
| 1500 | 38.00 | 0.025 | 478.30 | 0.319 | 775.70 | 0.517 | 1056.40 | 0.705 | 1417.10 | 0.945 |
| 1600 | 39.30 | 0.025 | 515.00 | 0.322 | 824.30 | 0.516 | 1137.20 | 0.711 | 1499.10 | 0.938 |
| 1700 | 38.50 | 0.023 | 539.30 | 0.317 | 872.00 | 0.513 | 1190.40 | 0.701 | 1604.00 | 0.944 |
| 1800 | 41.10 | 0.023 | 571.90 | 0.318 | 923.60 | 0.513 | 1263.80 | 0.703 | 1691.30 | 0.940 |
| 1900 | 41.40 | 0.022 | 605.60 | 0.319 | 978.60 | 0.515 | 1331.80 | 0.701 | 1786.30 | 0.941 |
| 2000 | 44.00 | 0.022 | 632.40 | 0.316 | 1023.80 | 0.512 | 1403.50 | 0.702 | 1883.90 | 0.942 |

As far as the parameterized *st*-orientation is concerned, we can extend our idea and insert more parameters $p_1, p_2, \dots, p_k$. In this case the algorithm would compute a longest path equal to $\Delta(p_1, p_2, \dots, p_k)$. These parameters will certainly define a choice on the structure that candidate sources are stored with more detail. For example, we can insert a parameter $k$ such that each time the $k$-th order statistic (or the median) from the timestamp vector is chosen.

The effectiveness of the parameterized *st*-orientation algorithm is fully indicated in the Experimental Results section.

## 4. Experimental results

Following we present our results for different kinds of graphs, *st*-Hamiltonian graphs, planar graphs and weighted graphs. All experiments were run on a Pentium IV machine, 512 MB RAM, 2.8 GHz under Windows 2000 professional.

### 4.1. *st*-Hamiltonian graphs

We have tested the parameterized STN algorithm for *st*-Hamiltonian graphs. In order to construct the graphs at random, we use the following algorithm. Initially, we compute a random permutation $P$ of the vertices of the graph. Then we construct a cycle by adding the undirected edges

$$(P(1), P(2)), (P(2), P(3)), \dots, (P(n-1), P(n)), (P(n), P(1))$$

and we choose at random two adjacent nodes of the cycle to be the source $s$ and the sink $t$ of our graph. This guarantees the existence of a Hamiltonian path from $s$ to $t$ and a possible maximum longest path of every *st*-oriented graph of length $n-1$.

Finally, we add the remaining $nd-n$ edges, given that the density of the desired graph is $d$. We keep a list of edges that have not been inserted and make exactly $nd-n$ random choices on this list, by simultaneously inserting the chosen undirected edge into the graph and updating the list of the remaining undirected edges. During the execution of the algorithm, ties between the timestamps of the candidate sources are broken at random. We isolate the nodes that satisfy the current timestamp condition (i.e., the nodes with maximum timestamp in case of maxSTN and the nodes with minimum timestamp in case of minSTN), and afterwards we choose a node from the isolated set at random. The algorithm was implemented in Java, using the Java Data Structures Library (www.jdsl.org) [25]. The graphs we have tested are $n$ node-undirected *st*-Hamiltonian graphs of density $d$ where $n = 100, 200, 300, \dots, 2000$ and $d = 3.5, 4.5, 5.5$. For each pair $(n, d)$ we have tested 10 different randomly generated graphs (and we present the mean of the length of the longest path) in order to get more reliable results. We have similar results for all other densities as well. In Tables 1–3 and in Fig. 9 our experimental results for the value of the parameter $p = 0, 0.3, 0.5, 0.7, 1$ are presented. Note the remarkable consistency of the longest path lengths with the parameter $p$.

### 4.2. Planar graphs

In this section we present our results for planar graphs. We have actually tested two classes of planar graphs (low density and triangulated planar graphs), and finally verified that the parameter works in a very efficient way for this class of graphs as well.

**Table 2**
Results for parameterized $st$-orientations of density 4.5 $st$-Hamiltonian graphs

| $n$ | $p=0$ | | $p=0.3$ | | $p=0.5$ | | $p=0.7$ | | $p=1$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ |
| 100 | 13.40 | 0.135 | 40.60 | 0.410 | 59.60 | 0.602 | 76.90 | 0.777 | 94.20 | 0.952 |
| 200 | 18.90 | 0.095 | 72.70 | 0.365 | 110.90 | 0.557 | 147.80 | 0.743 | 188.50 | 0.947 |
| 300 | 20.20 | 0.068 | 105.70 | 0.354 | 163.40 | 0.546 | 219.10 | 0.733 | 285.10 | 0.954 |
| 400 | 23.40 | 0.059 | 138.10 | 0.346 | 215.50 | 0.540 | 290.40 | 0.728 | 379.20 | 0.950 |
| 500 | 23.50 | 0.047 | 170.10 | 0.341 | 267.10 | 0.535 | 361.50 | 0.724 | 475.50 | 0.953 |
| 600 | 25.30 | 0.042 | 201.30 | 0.336 | 317.90 | 0.531 | 432.60 | 0.722 | 568.30 | 0.949 |
| 700 | 28.80 | 0.041 | 232.40 | 0.332 | 369.00 | 0.528 | 505.10 | 0.723 | 669.70 | 0.958 |
| 800 | 28.80 | 0.036 | 261.60 | 0.327 | 419.70 | 0.525 | 570.40 | 0.714 | 758.60 | 0.949 |
| 900 | 31.20 | 0.035 | 294.10 | 0.327 | 473.00 | 0.526 | 643.40 | 0.716 | 855.70 | 0.952 |
| 1000 | 30.60 | 0.031 | 321.00 | 0.321 | 521.50 | 0.522 | 713.80 | 0.715 | 952.40 | 0.953 |
| 1100 | 33.70 | 0.031 | 353.60 | 0.322 | 570.10 | 0.519 | 783.80 | 0.713 | 1051.50 | 0.957 |
| 1200 | 33.40 | 0.028 | 388.30 | 0.324 | 622.40 | 0.519 | 853.40 | 0.712 | 1141.40 | 0.952 |
| 1300 | 33.70 | 0.026 | 417.00 | 0.321 | 676.30 | 0.521 | 922.10 | 0.710 | 1236.50 | 0.952 |
| 1400 | 32.70 | 0.023 | 446.30 | 0.319 | 723.60 | 0.517 | 991.40 | 0.709 | 1335.80 | 0.955 |
| 1500 | 35.20 | 0.023 | 477.50 | 0.319 | 769.30 | 0.513 | 1061.60 | 0.708 | 1423.90 | 0.950 |
| 1600 | 37.30 | 0.023 | 512.00 | 0.320 | 825.00 | 0.516 | 1137.00 | 0.711 | 1523.10 | 0.953 |
| 1700 | 38.50 | 0.023 | 541.20 | 0.319 | 876.70 | 0.516 | 1199.30 | 0.706 | 1617.50 | 0.952 |
| 1800 | 38.30 | 0.021 | 567.10 | 0.315 | 929.40 | 0.517 | 1274.20 | 0.708 | 1709.40 | 0.950 |
| 1900 | 36.50 | 0.019 | 601.20 | 0.317 | 978.30 | 0.515 | 1340.30 | 0.706 | 1812.30 | 0.954 |
| 2000 | 40.60 | 0.020 | 632.70 | 0.317 | 1030.40 | 0.515 | 1410.40 | 0.706 | 1903.90 | 0.952 |

**Table 3**
Results for parameterized $st$-orientations of density 5.5 $st$-Hamiltonian graphs

| $n$ | $p=0$ | | $p=0.3$ | | $p=0.5$ | | $p=0.7$ | | $p=1$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ | $l(t)$ | $\frac{l(t)}{n-1}$ |
| 100 | 14.70 | 0.148 | 40.50 | 0.409 | 59.10 | 0.597 | 76.50 | 0.773 | 95.90 | 0.969 |
| 200 | 17.80 | 0.089 | 72.20 | 0.363 | 111.00 | 0.558 | 149.30 | 0.750 | 189.50 | 0.952 |
| 300 | 19.10 | 0.064 | 106.40 | 0.356 | 163.60 | 0.547 | 219.80 | 0.735 | 288.20 | 0.964 |
| 400 | 22.50 | 0.056 | 137.00 | 0.343 | 214.40 | 0.537 | 290.60 | 0.728 | 383.40 | 0.961 |
| 500 | 22.40 | 0.045 | 169.60 | 0.340 | 266.30 | 0.534 | 363.30 | 0.728 | 479.90 | 0.962 |
| 600 | 23.90 | 0.040 | 199.30 | 0.333 | 319.20 | 0.533 | 433.00 | 0.723 | 574.90 | 0.960 |
| 700 | 24.70 | 0.035 | 230.10 | 0.329 | 367.70 | 0.526 | 503.00 | 0.720 | 667.10 | 0.954 |
| 800 | 25.40 | 0.032 | 264.00 | 0.330 | 419.50 | 0.525 | 574.90 | 0.720 | 768.30 | 0.962 |
| 900 | 28.10 | 0.031 | 290.30 | 0.323 | 472.10 | 0.525 | 642.60 | 0.715 | 865.40 | 0.963 |
| 1000 | 30.10 | 0.030 | 323.60 | 0.324 | 518.80 | 0.519 | 716.30 | 0.717 | 958.20 | 0.959 |
| 1100 | 34.20 | 0.031 | 352.20 | 0.320 | 572.90 | 0.521 | 784.20 | 0.714 | 1053.30 | 0.958 |
| 1200 | 33.20 | 0.028 | 385.50 | 0.322 | 625.00 | 0.521 | 854.20 | 0.712 | 1152.40 | 0.961 |
| 1300 | 31.60 | 0.024 | 417.20 | 0.321 | 673.70 | 0.519 | 923.80 | 0.711 | 1245.60 | 0.959 |
| 1400 | 31.10 | 0.022 | 446.60 | 0.319 | 724.70 | 0.518 | 995.90 | 0.712 | 1343.00 | 0.960 |
| 1500 | 34.20 | 0.023 | 479.30 | 0.320 | 776.00 | 0.518 | 1067.10 | 0.712 | 1442.70 | 0.962 |
| 1600 | 35.70 | 0.022 | 507.40 | 0.317 | 825.50 | 0.516 | 1138.60 | 0.712 | 1531.50 | 0.958 |
| 1700 | 34.00 | 0.020 | 537.60 | 0.316 | 879.30 | 0.518 | 1207.40 | 0.711 | 1631.00 | 0.960 |
| 1800 | 40.40 | 0.022 | 567.70 | 0.316 | 926.30 | 0.515 | 1278.80 | 0.711 | 1728.20 | 0.961 |
| 1900 | 37.30 | 0.020 | 597.40 | 0.315 | 980.80 | 0.516 | 1346.10 | 0.709 | 1827.80 | 0.963 |
| 2000 | 37.30 | 0.019 | 632.80 | 0.317 | 1027.10 | 0.514 | 1413.70 | 0.707 | 1920.20 | 0.961 |

Low density (roughly equal to 1.5) $st$-planar graphs are constructed as follows: We build up a tree of $n$ nodes by randomly picking up a node and setting it to be the root of the tree. Then we connect the current tree (initially it only consists of the root) with a node that does not belong to the current tree, and which is chosen at random. We execute the same procedure until all nodes are inserted into the tree. Then we connect the leaves of the tree following a preorder numbering, so that all crossings are avoided. In Table 4 the results for this class of graphs are shown. Note that the effect of the parameter is again evident.

Maximum density ($m = 3n - 6$) $st$-planar graphs were computed with a certain software for graph algorithms and visualization called P.I.G.A.L.E.[2] This software produces graphs in ascii format, which are easily transformed to an input for our algorithm. From Table 5, we can see that the parameter $p$ actually *defines* the length of the longest path for triangulated planar graphs as well.

[2] Public Implementation of a Graph Algorithm Library and Editor (http://pigale.sourceforge.net/).

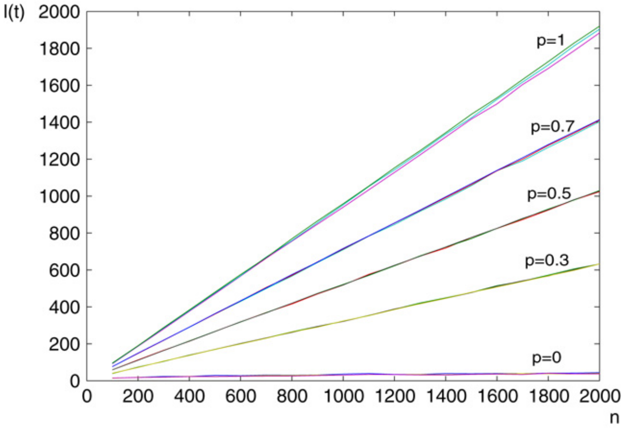



**Fig. 9.** Parameterized longest path length results.

**Table 4**
Results for low density planar graphs

| $n$ | $p=0$ | $p=0.5$ | $p=1$ |
|---|---|---|---|
| | $l(t)$ | $l(t)$ | $l(t)$ |
| 250 | 123.10 | 168.90 | 216.90 |
| 500 | 229.50 | 297.40 | 399.60 |
| 750 | 360.10 | 489.40 | 629.10 |
| 1000 | 485.20 | 639.60 | 831.40 |
| 1250 | 592.30 | 818.00 | 1060.70 |
| 1500 | 651.00 | 991.60 | 1304.10 |
| 1750 | 842.10 | 1145.70 | 1486.30 |
| 2000 | 910.30 | 1302.80 | 1686.10 |
| 2250 | 1077.20 | 1448.40 | 1892.60 |
| 2500 | 1134.10 | 1539.80 | 2053.50 |
| 2750 | 1350.70 | 1700.70 | 2198.10 |
| 3000 | 1451.30 | 2025.80 | 2590.20 |
| 3250 | 1418.80 | 2156.00 | 2814.40 |

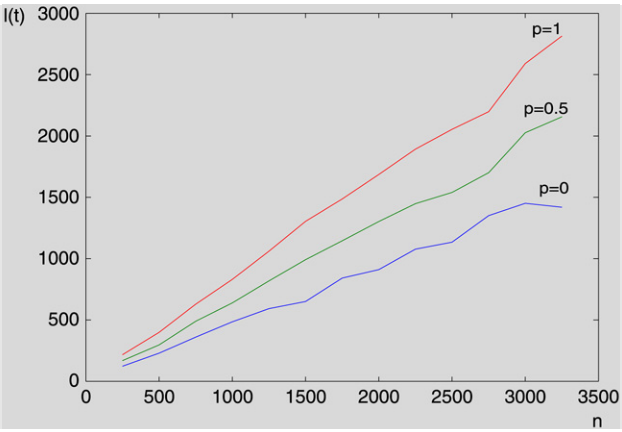
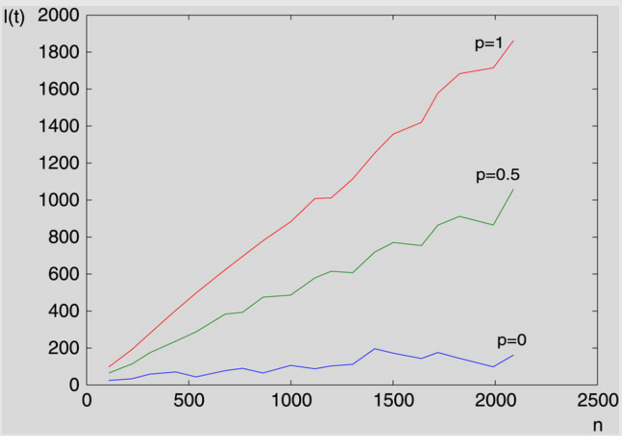



**Table 5**
Results for triangulated planar graphs

| $n$ | $p=0$ | $p=0.5$ | $p=1$ |
|---|---|---|---|
| | $l(t)$ | $l(t)$ | $l(t)$ |
| 109 | 25.00 | 65.00 | 98.00 |
| 222 | 34.00 | 114.00 | 192.00 |
| 310 | 59.00 | 175.00 | 280.00 |
| 436 | 71.00 | 237.00 | 404.00 |
| 535 | 44.00 | 287.00 | 497.00 |
| 678 | 78.00 | 383.00 | 623.00 |
| 763 | 90.00 | 393.00 | 695.00 |
| 863 | 65.00 | 475.00 | 780.00 |
| 998 | 106.00 | 486.00 | 882.00 |
| 1117 | 88.00 | 579.00 | 1008.00 |
| 1197 | 103.00 | 615.00 | 1012.00 |
| 1302 | 112.00 | 607.00 | 1114.00 |
| 1410 | 196.00 | 719.00 | 1254.00 |
| 1501 | 172.00 | 771.00 | 1357.00 |
| 1638 | 143.00 | 754.00 | 1420.00 |
| 1719 | 176.00 | 864.00 | 1578.00 |
| 1825 | 144.00 | 912.00 | 1683.00 |
| 1990 | 98.00 | 865.00 | 1715.00 |
| 2089 | 162.00 | 1059.00 | 1862.00 |



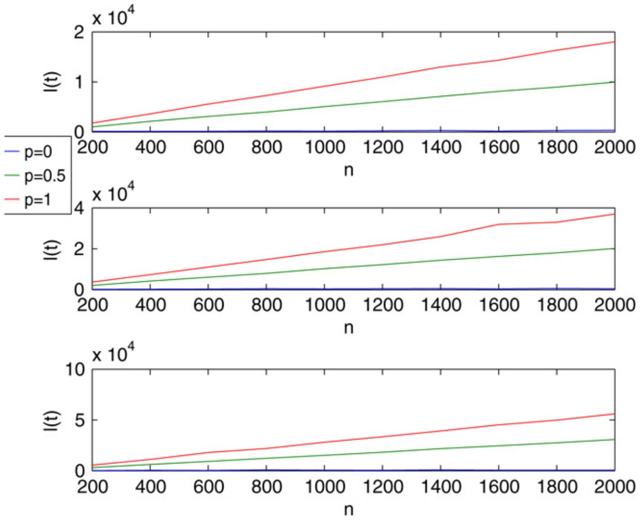



**Fig. 10.** Results for weighted graphs for $W = 10, 20, 30$ (up to bottom).

### 4.3. Weighted graphs

Finally, the third series of experiments were conducted on weighted graphs (Fig. 10). We used the algorithm described in Section 4.3 and make use of the parameter $p$ in the same way as in the case of undirected graphs. The weighted graphs were constructed as follows. Firstly we construct a respective $st$-Hamiltonian unweighted graph. Then we set a value $W$ to be an upper bound on the weights of the edges of the graph. We set the weights of the edges that lie on a Hamiltonian path from $s$ to $t$ equal to $W$. Clearly, the maximum longest path length of an $st$-orientation that corresponds to such weighted graphs is $(n-1)W$. The weights of the remaining edges are uniformly distributed in $[1, W]$. Note that the length of the longest path of the $st$-orientation, in this case, is in absolute accordance with the value of the parameter $p$ and the value $W$ (Fig. 10).

## 5. Conclusions

In this paper, a new algorithm for computing $st$-orientations of graphs is presented. The novelty of the algorithm lies in the fact that it gives us the opportunity to control some characteristics of the final $st$-oriented graph, such as the length of the longest path. Many of the applications which use an $st$-numbering as a first step can use this algorithm in order to produce better solutions. Experimental results for various classes of graphs reveal the robustness of the algorithm. Future work includes:

- Further theoretical results concerning the properties of the algorithm.
- Applications of parameterized $st$-Orientations in Graph Drawing (Hierarchical Drawing, Visibility Representations and Orthogonal Drawings). Can we achieve better area bounds?
- Graph Coloring via minSTN and longest path via maxSTN. Can we produce good heuristics for these problems?
- Constrained $st$-Orientations. How can we modify the developed algorithm to produce parameterized $st$-orientations with some predefined orientations on edges (we conjecture this problem is NP-hard)?
- Cheaper update of the block-cutpoint tree using sophisticated data structures such as [21].
- How does the parameterized primal longest path length influence the dual longest path length in planar graphs?
- Can we prove that the developed algorithm *may* reach any possible $st$-orientation that corresponds to a certain graph?


## Acknowledgements

The authors would like to thank Franco P. Preparata and Roberto Tamassia for useful comments and discussions. They thank Philip Klein for indicating the $O(\log n)$ biconnectivity maintainance for planar graphs. Finally they thank the anonymous reviewers for their very useful comments.

This work was performed while the first author was with the University of Crete and ICS-FORTH.


## References


[1] C. Papamanthou, I.G. Tollis, Parameterized $st$-orientations of graphs: algorithms and experiments, in: Proc. International Conference on Graph Drawing, 2006, pp. 220–233.

[2] R. Tamassia, I.G. Tollis, A unified approach to visibility representations of planar graphs, Discrete and Computational Geometry 1 (1986) 321–341.

[3] A. Papakostas, I.G. Tollis, Algorithms for area-efficient orthogonal drawings, Computational Geometry: Theory and Applications 9 (1998) 83–110.
[4] G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis, Annotated bibliography on graph drawing algorithms, Computational Geometry: Theory and Applications 4 (1994) 235–282.
[5] G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis, Graph Drawing: Algorithms for the Visualization of Graphs, Prentice Hall, 1999.
[6] K. Sugiyama, S. Tagawa, M. Toda, Methods for visual understanding of hierarchical systems, Transactions on Systems, Man, and Cybernetics SMC-11 (2) (1981) 109–125.
[7] F. Annexstein, K. Berman, Directional routing via generalized *st*-numberings, Discrete Mathematics 13 (2) (2000) 268–279.
[8] M. Mursalin Akon, S. Asaduzzaman, Md. Saidur Rahman, M. Matsumoto, Proposal for *st*-routing, Telecommunication Systems 25 (3–4) (2004) 287–298.
[9] S. Nakano, Md. Saidur Rahman, T. Nishizeki, A linear-time algorithm for four-partitioning four-connected planar graphs, Information Processing Letters 62 (1997) 315–322.
[10] S. Even, R. Tarjan, Computing an *st*-numbering, Theoretical Computer Science 2 (1976) 339–344.
[11] C. Papamanthou, I.G. Tollis, Applications of parameterized *st*-orientations in graph drawing algorithms, in: Proc. International Conference on Graph Drawing, 2005, pp. 355–367.
[12] A. Lempel, S. Even, I. Cederbaum, An algorithm for planarity testing of graphs, in: Theory of Graphs: International Symposium, 1967, pp. 215–232.
[13] J. Ebert, *st*-ordering the vertices of biconnected graphs, Computing 30 (1) (1983) 19–33.
[14] R. Tarjan, Two streamlined depth-first search algorithms, Fundamenta Informaticae 9 (1986) 85–94.
[15] P. Rosenstiehl, R. Tarjan, Rectilinear planar layout and bipolar orientation of planar graphs, Discrete Computational Geometry 1 (1986) 343–353.
[16] Y. Maon, B. Schieber, U. Vishkin, Parallel ear decomposition search (eds) and *st*-numbering in graphs, Theoretical Computer Science 47 (1986) 277–298.
[17] Ulrik Brandes, Eager *st*-ordering, in: Proc. European Symposium on Algorithms, 2002, pp. 247–256.
[18] H. De Fraysseix, P. Ossona de Mendez, P. Rosenstiehl, Bipolar orientations revisited, Discrete Applied Mathematics 56 (1995) 157–179.
[19] C. Papamanthou, I.G. Tollis, *st*-numberings and longest paths, manuscript, ICS-FORTH, 2004.
[20] J. Hopcroft, R. Tarjan, Efficient algorithms for graph manipulation, Communications of ACM 16 (1973) 372–378.
[21] J. Holm, K. de Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity, Journal of the ACM 48 (4) (2001) 723–760.
[22] T. Lukovszki, W. Strothmann, Decremental biconnectivity on planar graphs, Technical report, University of Paderborn, tr-ri-97-186, 1997.
[23] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, W.H. Freeman and Company, 1979.
[24] T. Gallai, On directed paths and circuits, in: Theory of Graphs: International Symposium, 1968, pp. 215–232.
[25] Michael T. Goodrich, Roberto Tamassia, Data Structures and Algorithms in Java, fourth ed., John Wiley & Sons, 2005.