Parameterized *st*-Orientations of Graphs: Algorithms and Experiments

Charalampos Papamanthou[1] and Ioannis G. Tollis[2]

[1] Department of Computer Science, Brown University, P.O. Box 1910, Providence RI, U.S.A.
cpap@cs.brown.edu
[2] Department of Computer Science, University of Crete, P.O. Box 2208 & Institute of Computer Science, FORTH, P.O. Box 1385 Heraklion, Greece
tollis@ics.forth.gr

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. However, as indicated in [1], the computation of more than one *st*-orientation is very important for many applications in multiple research areas, such as this of Graph Drawing. In this paper we show how to compute such orientations with certain (parameterized) characteristics in the final *st*-oriented graph, such as the length of the longest path. Apart from Graph Drawing, this work applies in other areas such as Network Routing and in tackling difficult problems such as Graph Coloring and Longest Path. We present primary approaches to the problem of computing longest path parameterized *st*-orientations of graphs, an analytical presentation (together with proof of correctness) of a new $O(m \log^5 n)$ ($O(m \log n)$ for planar graphs) time algorithm that computes such orientations (and which was used in [1]) and extensive computational results that reveal the robustness of the algorithm.

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 [2,3,4,5,6], network routing [7,8] and graph partitioning [9].

st-numberings were first introduced in 1967 in [10], 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 [10] 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 [11]. Ebert [12] presented a slightly simpler algorithm for the computation of such a numbering, which was further simplified by Tarjan [13]. The planar

M. Kaufmann and D. Wagner (Eds.): GD 2006, LNCS 4372, pp. 220–233, 2007.

case has been extensively investigated in [14] where a linear time algorithm is presented which may reach any *st*-orientation of a planar graph. Additionally, in [15] a parallel algorithm is described (running in $O(\log n)$ time using $O(m)$ processors) and finally in [16] another linear time algorithm for the problem is presented. An overview of the work concerning bipolar orientations is presented in [17].

However, as indicated in [1], the computation of more than one *st*-orientation is very important for many applications in the area of Graph Drawing. Most algorithms use any algorithm that produces such an orientation, e.g., [11], without expecting any specific properties of the oriented graph.

In this paper we approach the problem of computing *st*-orientations of specific properties. We present and give proof of correctness of algorithms that are able to control the length of the longest path of the resulting directed acyclic graph. The algorithms run in $O(m \log^5 n)$ time ($O(m \log n)$ time for planar graphs). This provides significant flexibility to many graph algorithms and applications [2,3,7,8,9]. We also present extensive experimental results that reveal the robustness of the algorithm.

2 Preliminaries

2.1 The Very First Approach

The aim of this work has always been the computation of *st*-orientations of longest path length that can be efficiently controlled by an input. 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 [11], proceed by choosing over a set a vertices. Thus in order to produce multiple *st*-orientations using these algorithms, one should try to consider different combinations of successive vertices. Some heuristics applied for the Tarjan-Even algorithm are described in [18], where after extensive computational results, we reached to the conclusion that exploiting the *freedom* of choice of successive vertices this algorithm gives us, can only lead to *st*-orientations that almost have **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. As a result of this, we seeked for new algorithms that achieve this goal.

2.2 Exploiting Biconnectivity

The main idea behind the algorithm was the explosion of the biconnectivity structure of a graph. This finally seemed to be of great importance in a way that we could compute both a correct *st*-orientation of a graph and we could efficiently influence the length of the longest path of the computed *st*-orientation by using some information this structure can provide.

Following, we introduce some terminology and preliminary results. Throughout the paper, $N_G(v)$ denotes the set of neighbors of node v in graph G, s the source of the graph, t the sink of the graph and $l(u)$ the length of the longest path of a node u from the source s of the 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 [19] 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.

The block-cutpoint tree is a free tree, i.e., it has no distinguished 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.

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 [19]. Following, we give the description of the algorithm.

3 The Algorithm

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 removed 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 orient 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

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 resulting st-oriented graph.

Lemma 1. *Let $G = (V, E)$ be an 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 of the t-rooted block-cutpoint tree of $G - \{s\}$. Moreover, this neighbor is not a cutpoint.*

Proof. If graph $G - \{s\}$ is still biconnencted, 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, suppose that there is a leaf-block ℓ 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, 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 t-rooted block-cutpoint tree, which is not a cutpoint. □

Let now $G = (V, E)$ 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 $i = 1, \ldots, n-1$ (note that the subscript i of v_i denotes the order with which the nodes are removed).

During the procedure (which we call STN) 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 next source. Q initially contains the desired source for the final orientation, s. Finally we maintain the leaf-blocks of the t-rooted block-cutpoint tree. During every iteration i of the algorithm node v_i is chosen so that

- it is a non-cutpoint node that is an element of 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 nodes $u \in N_{G_i}(v_i)$ 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 gets 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 2. *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 $u \rightarrow v$. 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 single node). But all nodes u are chosen to be neighbors of prior sources. By Lemma 1, u can never be a cutpoint, hence node $v \neq t$ will certainly become a source exactly once. Finally, Q gets empty at the end of the algorithm as each time at least one node is added into Q and exactly one node is removed from it. □

By Lemmas 1, 2, we see that at each iteration of the algorithm there will be at least one node to be chosen as a future source.

Corollary 3. *Suppose after a vertex v 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 4. *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 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 inserted 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 one sink. □

Lemma 5. *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 4, 5 we have:

Theorem 6. *The directed graph $F = (V', E')$ is st-oriented.* □

Alg.1 is a recursive algorithm for the *st*-orientation computation of a biconnected undirected graph G. 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

Algorithm 1. STN(G, s, t)

```
1: Initialize F = (V', E');
2: Initialize m(i) = 0 for all nodes i of the graph; (timestamp vector)
3: j = 0; {Initialize a counter}
4: Q = {s}; {Insert s into Q}
5: STREC(G, s); {Call the recursive algorithm}
6: ------------------------------------------
7: function STREC(G, v)
8: j = j + 1;
9: f(v) = j;
10: V = V − {v};   {A source is removed from G}
11: V' = V' ∪ {v};  {and is added to F}
12: for all edges (v, i) ∈ E do
13:    E = E − {(v, i)}
14:    E' = E' ∪ {(v, i)}
15: end for
16: Q = Q ∪ {N(v) ∼ t} − {v}; {The set of possible next sources}
17: m(N(v)) = j;
18: if Q == {∅} then
19:    f(t) = n;
20:    return;
21: else
22:    T(t, B_j^1, B_j^2, ..., B_j^r)=UpdateBlocks(G); {Update the block-cutpoint tree; h_j^i is
       the cutpoint that defines the leaf-block B_j^i}
23:    for all leaf-blocks (B_j^i, h_j^i) do
24:       choose v_ℓ ∈ B_j^ℓ ∩ Q ∼ {h_j^ℓ}
25:       STREC(G, v_ℓ);
26:    end for
27: end if
```

removed from the graph, the subscript i is the final st-number of node v_i. The st-numbering can also be 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 are 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 became candidate for removal.

Note that the recursion is executed exactly $n - 1$ times. The running time of each recursive call is consumed by the procedure that updates the block-cutpoint tree, which is $O(n + m)$ [19]. 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 [20] 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, we obtain the following:

Theorem 7. *Algorithm STN can be implemented to run in* $O(m \log^5 n)$ *time.* □

Finally, for planar graphs, we can compute biconnected components in $O(\log n)$ amortized time per edge deletion due to [21]. Hence, the algorithm can be implemented to run in $O(m \log n)$ time for planar graphs [1].

Finally, the *st*-orientation algorithm defines an *st-tree* T_s (Figure 1). 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. 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.

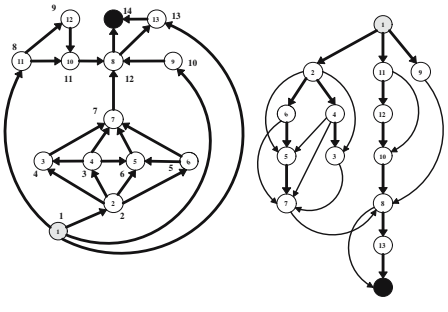


Fig. 1. Algorithm execution on a graph (left) and the respective *st*-tree (right). Beside each node of the graph the **STN** rank of visit is depicted.

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 8. *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* $l(t) \geq d(T_s)$. □

Additionally, there is a strong connection between the structure of the t-rooted block-cutpoint tree and the length of the longest path of the *st*-oriented graph that STN computes:

[1] We thank Philip Klein for this observation.

Theorem 9. *Suppose STN is run on an undirected st-Hamiltonian graph*[2] 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* $i = 1, 2, \ldots, n-1$. *Then* $l(t) \leq n - 1 - \sum_{k_i > k_{i-1}} (k_i - k_{i-1})$. □

4 Longest Path Parameterized Orientations

4.1 Maximal and Minimal Case

We have used two approaches in order to produce st-oriented graphs with long longest path and st-oriented graphs with small longest path. As presented in Section 3, during each iteration of the algorithm a timer j (line 8 of Algorithm 1) 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 higher probability to obtain a long longest path length. We call this way of choosing vertices MAX-STN. Actually, MAX-STN resembles a DFS traversal (it searches the graph at a *maximal* depth). Hence, during MAX-STN, 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 length. We call this way of choosing vertices MIN-STN, which in turn resembles a BFS traversal. Hence, during MIN-STN, the next source v is arbitrarily chosen from the set $\{v \in Q' : m(v) = \min\{m(i) : i \in Q'\}\}$. Note that the choice of the new vertex with the minimum or maximum timestamp does not influence the running time of the algorithm (it can be done in $O(\log n)$ time) since we can implement Q' as a priority queue.

4.2 Medium (Parameterized) Case

Instead of computing st-oriented graphs with either long or small length of longest path, it would be desirable to be able to produce medium (parameterized) longest path st-oriented graphs. 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 a directed acyclic graph of length of longest path that is a function of p?

It turns out that 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 MAX-STN for the first pn iterations and MIN-STN for the remaining $(1-p)n$ iterations. We call this method PAR-STN(p) and we say that it produces an st-oriented graph with length of longest path from s to t equal to $\Delta(p)$. Note that PAR-STN(0) is equivalent to MIN-STN and $\Delta(0) = \lambda(t)$, while PAR-STN(1)

[2] We say that a graph is st-Hamiltonian when it has at least one simple path from s to t that contains all the other nodes of the graph.

is equivalent to MAX-STN and $\Delta(1) = \ell(t)$. PAR-STN has been tested and it produces longest paths with $\Delta(p) \geq p(n-1)$, when applied to *st*-Hamiltonian graphs. Actually, $\Delta(p)$ is very close to $p(n-1)$. Additionally, it has been observed that if we switch the order of MAX-STN and MIN-STN execution, i.e., execute MIN-STN for the first pn iterations and MAX-STN for the remaining $(1-p)n$ iterations, then we get longest paths with $\Delta(p) \leq p(n-1)$. These observations are fully clarified in the experimental results, where it is evident that the algorithm can compute many *st*-numberings within $[\lambda(t), \ell(t)]$. In this way, applications that use *st*-numberings can use this interval to compute an optimized solution.

4.3 Longest Path Timestamps and Weighted Graphs

If we apply a relaxation phase during STN, we can compute the longest path length $l(v)$ from s to every node v during the execution of STN. This can be achieved as follows: At the beginning, we initialize the longest path vector l to be the zero vector, hence $l(v) = 0\ \forall v \in V$. Suppose that at a random iteration of the algorithm we remove a node u and we orient all u's incident edges (u, i) away from u. For every oriented edge $(u, i) \in E'$ of weight w_{ui} (in case of unweighted graphs it is $w_{ui} = 1$) we relax $l(i)$ as follows:

1: **for all** $(u, i) \in E'$ **do**
2: **if** $l(i) < l(u) + w_{ui}$ **then**
3: $l(i) = l(u) + w_{ui}$;
4: **end if**
5: **end for**

Instead of now using the timestamps $m(u)$ to choose the next source of the algorithm, we can use the *online* computed longest paths $l(u)$.

This method mainly applies to the case of weighted graphs. By using the *relaxed* longest path length as a timestamp, we can produce long or short *st*-orientations of weighted graphs. Hence, 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.

5 Some Applications

There are many areas where parameterized *st*-orientations can apply. For example, many Graph Drawing Algorithms [2,3,4,5,6] use an *st*-orientation as their first step. Actually, the length of the longest path of the used *st*-orientation determines the area bounds of the final drawing (the area can be reduced by a factor of n for some classes of graphs [1]). More details can be found in [1].

Additionally, MAX-STN can be used as a heuristic for the longest path problem in undirected graphs. Actually, as we will see in the experimental results section, MAX-STN produces *near* optimal results for this problem.

MIN-STN can even be used to compute good colorings of graphs. By producing a good solution for the minimum *st*-orientation problem we can obtain a

good solution for the graph coloring problem, based on the polynomial reduction between the two problems [22].:

Theorem 10. *([22]) Let $G = (V, E)$ be a connected graph and let $G' = (V \cup \{s, t\}, \{E \cup \{s, i\} \cup \{t, i\} \forall i \in V\})$. χ is the chromatic number of G if and only if G' can be st-oriented with a minimum longest path length st-orientation with $l(t) = \chi + 1$.* □

Based on the above theorem (which actually justifies the NP-hardness of the minimum longest path length st-orientation problem[3]), we used PAR-STN(0) to st-orient G' and derive a coloring for G. We give some indicative results for the graph coloring problem in the experimental results section.

6 Experimental Results

The first tests were conducted on st-Hamiltonian Graphs (Table 1). We used this class of graphs as they have an a priori known upper bound for the maximum longest path length equal to $n - 1$. In this way, we could see how effective the parameter p is. In order to construct the graphs in random, we compute a random permutation P of the vertices of the graph. Then we construct a cycle by adding the undirected edges $(P(1), P(2)), (P(2), P(3)), \ldots, (P(n - 1), P(n)), (P(n), P(1))$ and we chose 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 length 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 MAX-STN and the nodes with minimum timestamp in case of MIN-STN) and afterwards we choose a node from the isolated set at random.

The second series of experiments was conducted on low density (roughly equal to 1.5) planar graphs (Table 2). These graphs were 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 till 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.

Finally, the third series of experiments were conducted on weighted graphs (Figure 2). We used the algorithm described section 4.3 and make use of

[3] Note that the maximum longest path length st-orientation problem is NP-hard by reduction from the directed Hamilton Path problem.

Table 1. Results for density 6.5 *st*-Hamiltonian graphs

n	p=0 $\frac{l(t)}{(n-1)}$	p=0.3 $\frac{l(t)}{(n-1)}$	p=0.5 $\frac{l(t)}{(n-1)}$	p=0.7 $\frac{l(t)}{(n-1)}$	p=1 $\frac{l(t)}{(n-1)}$
200	0.085	0.372	0.568	0.743	0.963
400	0.051	0.341	0.540	0.731	0.964
600	0.041	0.332	0.532	0.726	0.963
800	0.033	0.330	0.527	0.721	0.962
1000	0.027	0.325	0.521	0.716	0.967
1200	0.024	0.322	0.521	0.718	0.965
1400	0.024	0.318	0.515	0.714	0.964
1600	0.020	0.318	0.515	0.712	0.964
1800	0.020	0.315	0.514	0.710	0.966
2000	0.019	0.314	0.514	0.710	0.964

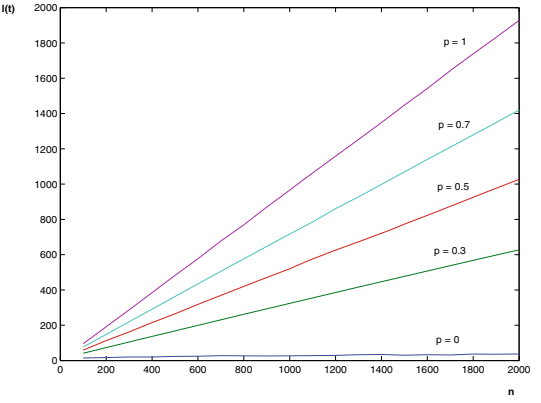
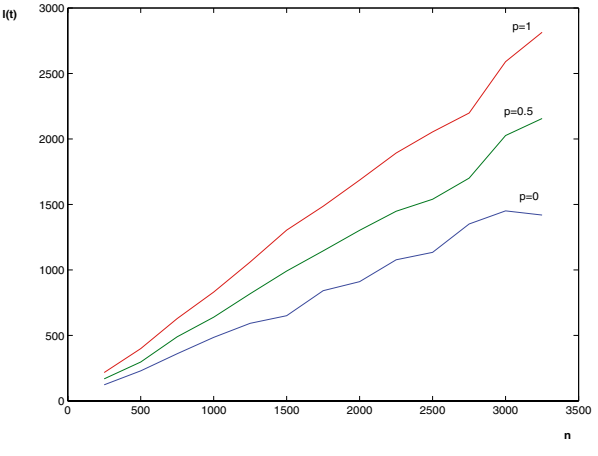


Table 2. Results for low density planar graphs

n	p=0 $l(t)$	p=0.5 $l(t)$	p=1 $l(t)$
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



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 $[1, W]$.

From Tables 1 and 2 we can see that, by using parameter p, we can influence the length of the longest path of the final *st*-oriented graph. It is remarkable that for the class of *st*-Hamiltonian graphs (Table 1) the computed length of longest path for a parameter p is approximately $p(n-1)$. For the class of low density

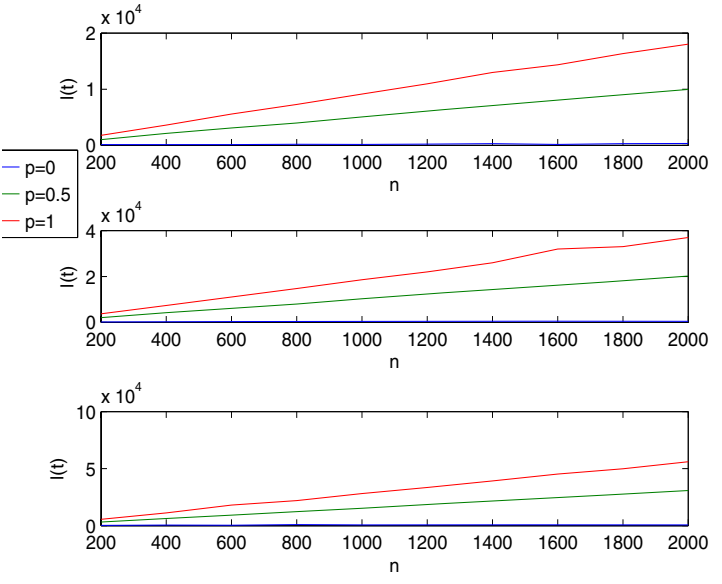


Fig. 2. Results for weighted graphs for $W = 10, 20, 30$ (up to bottom)

planar graphs (Table 2), the algorithm performs very well and indeed computes different *st*-orientations according to the parameter. Finally, for the class of the weighted graphs, note that the length of the longest path of the *st*-orientation is in absolute accordance with the value of the parameter p and the value W. More resutls can be found at `http://www.csd.uoc.gr/~cpap/MSc_thesis.ps`.

As far as the applications of parameterized *st*-orientations in graph coloring are concerned, we have tested known benchmarks available at `http://mat.gsia.cmu.edu/COLOR/instances.html` (Table 3) and got good

Table 3. Some benchmark graphs for which MIN-STN has computed an optimal or near optimal coloring

file name	n	m	optimal coloring	MIN-STN (p=0) coloring
games120.col	120	368	9	9
jean.col	80	254	10	10
huck.col	74	301	11	11
zeroin.i.1.col	211	4100	49	49
mulsol.i.3.col	184	3916	31	31
mulsol.i.1.col	197	3925	49	49
fpsol2.i.1.col	496	11654	65	65
miles250.col	128	387	8	9
anna.col	138	493	11	12
inithx.i.2.col	645	13979	31	32

results for most of the tested graphs. Results are promising for other classes of graphs also.

7 Conclusions and Future Work

In this paper, we show that there is a very efficient way to control the length of the longest path of a produced *st*-orientation. In this way, we are able to produce more than one *st*-numberings of certain quality. This work is especially important from an applied point of view. The parameterized *st*-orientations can be used by various algorithms that use an *st*-numbering as their first step. In this way, better solutions to certain problems can be produced [1]. Concerning future work, some interesting questions that come out of this work are the following:(a) Can we prove that the presented algorithm may reach **any** possible *st*-orientation (note that the algorithm can also choose non-cutpoint nodes that belong to *some* block of the block-cutpoint tree)? and (b) Can we compute an *st*-orientation given a set of edges that have a predefined orientation (constrained *st*-orientation problem)? (c) Implementation of efficient data structures for the *t*-rooted block-cutpoint tree.

Acknowledgements. The authors would like to thank Franco P. Preparata (Brown University), Roberto Tamassia (Brown University) and Mihalis Yannakakis (Columbia University) for useful comments and discussions.

References

1. C. Papamanthou and I.G. Tollis. Applications of parameterized *st*-orientations in graph drawing algorithms. In *LNCS Graph Drawing 2004*, volume 3843, pages 355–367, 2005.
2. R. Tamassia and I.G. Tollis. A unified approach to visibility representations of planar graphs. *Disc. and Comp. Geom.*, 1:321–341, 1986.
3. A. Papakostas and I.G. Tollis. Algorithms for area-efficient orthogonal drawings. *Computational Geometry: Theory and Applications*, 9:83–110, 1998.
4. G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis. Annotated bibliography on graph drawing algorithms. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.
5. G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.
6. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-11(2):109–125, 1981.
7. F. Annexstein and K. Berman. Directional routing via generalized st-numberings. *SIAM J. Discrete Math.*, 13(2):268–279, 2000.
8. M. Mursalin Akon, S. Asaduzzaman, M.S. Rahman, and M. Matsumoto. Proposal for st-routing. *Tellecommunication Systems*, 25(3-4):287–298, 2004.
9. S. Nakano, M.S. Rahman, and T. Nishizeki. A linear-time algorithm for four-partitioning four-connected planar graphs. *Information Processing Letters*, 62:315–322, 1997.

10. A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *P. Rosenstiehl(ed.) Theory of Graphs: Tihany, Academic Press, New York*, pages 115–118, 1968.
11. S. Even and R. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2:339–344, 1976.
12. J. Ebert. st-ordering the vertices of biconenected graphs. *Computing*, 30(1):19–33, 1983.
13. R. Tarjan. Two streamlined depth-first search algorithms. *Fundamentae Informatica*, 9:85–94, 1986.
14. P. Rosenstiehl and R. Tarjan. Rectilinear planar layout and bipolar orientation of planar graphs. *Discrete Comput. Geom.*, 1:343–353, 1986.
15. Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theoret. Comput. Sci*, 47:277–298, 1986.
16. Ulrik Brandes. Eager st-ordering. In *LNCS ESA 2002*, volume 2461, pages 247–256, 2002.
17. H.D. Fraysseix, P.O. de Mendez, and P. Rosenstiehl. Bipolar orientations revisited. *Discrete Applied Mathematics*, 56:157–179, 1995.
18. C. Papamanthou and I.G. Tollis. *st*-numberings and longest paths, *manuscript, ICS-FORTH 2004*.
19. J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *Comm. ACM*, 16:372–378, 1973.
20. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. *J. ACM*, 48(4):723–760, 2001.
21. Tams Lukovszki and Willy-B. Strothmann. Decremental biconnectivity on planar graphs technical report, university of paderborn, tr-ri-97-186.
22. T. Gallai. On directed paths and circuits. In *P. Erdos(ed.) Theory of Graphs: International Symposium July 1966*, pages 215–232, 1967.