COMPUTING LONGEST PATH PARAMETERIZED st-ORIENTATIONS OF GRAPHS: ALGORITHMS AND APPLICATIONS

By
Charalampos Papamanthou

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
UNIVERSITY OF CRETE
HERAKLION, GREECE
JULY 2005

UNIVERSITY OF CRETE
DEPARTMENT OF
COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**Computing Longest Path Parameterized** st**-Orientations of Graphs: Algorithms and Applications**" by **Charalampos Papamanthou** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: July 2005

Supervisor: ______________________
Ioannis G. Tollis, Professor

Readers: ______________________
George F. Georgakopoulos, Assistant Professor

____
Evangelos Markatos, Associate Professor

Approved by the Chairman of the Graduate Studies Committee:

____
Dimitris Plexousakis, Associate Professor

UNIVERSITY OF CRETE

Date: **July 2005**

Author: **Charalampos Papamanthou**

Title: **Computing Longest Path Parameterized st-Orientations of Graphs: Algorithms and Applications**

Department: **Computer Science**

Degree: **M.Sc.** Convocation: **July** Year: **2005**

Signature of Author

Στους γονείς μου Ιωάννη και Γεωργία,
και στον αδερφό μου Χρήστο

Table of Contents

List of Tables

List of Figures

Abstract

Computing Longest Path Parameterized *st*-Orientations of Graphs: Algorithms and Applications

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 (hierarchical drawings, visibility representations, orthogonal drawings), graph coloring, longest path and network routing. Most algorithms use any algorithm that produces such an orientation, without expecting any specific properties of the oriented graph.

In this thesis we present a new algorithm that computes *st*-orientations with certain characteristics. Actually, we describe new algorithms 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*-orientation. The importance of this research direction has been implied in the past, especially in the field of Graph Drawing.

Our algorithms are able to compute *st*-oriented graphs of "parameter-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 are dependent on the length of the longest path of the *st*-oriented graph. Moreover, certain *st*-orientations of graphs can approximate suitably formulated graph problems (longest path, graph coloring). Finally, 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. We investigate most of these applications and show that there is indeed a need for parameterized *st*-numberings.

Περίληψη

Υπολογισμός Παραμετρικών ως προς το Μακρύτερο Μονοπάτι st-Προσανατολισμών Γράφων: Αλγόριθμοι και Εφαρμογές

Το πρόβλημα του προσανατολισμού ενός μη κατευθυνόμενου γράφου έτσι ώστε να έχει μια πηγή (source), μια δεξαμενή (sink) και καθόλου κύκλους (st-προσανατολισμός), είναι πολύ σημαντικό σε πολλούς αλγόριθμους γράφων και εφαρμογές, όπως η σχεδίαση γράφων (ιεραρχική σχεδίαση, αναπαράσταση ορατότητας, ορθογώνια σχεδίαση), ο χρωματισμός γράφων, το πρόβλημα του μακρύτερου μονοπατιού και η δρομολόγηση δικτύων. Οι περισσότεροι αλγόριθμοι χρησιμοποιούν οποιονδήποτε αλγόριθμο που παράγει έναν τέτοιο προσανατολισμό χωρίς να απαιτούν κάποιες συγκεκριμμένες ιδιότητες του προσανατολισμένου γράφου.

Σε αυτήν την εργασία παρουσιάζεται ένας καινούριος αλγόριθμος που υπολογίζει st-προσανατολισμούς γράφων συγκεκριμένων χαρακτηριστικών. Συγκεκριμένα, περιγράφουμε νέους αλγορίθμους με θεωρητικά και πειραματικά αποτελέσματα που δείχνουν ότι υπάρχει ένας αποδοτικός τρόπος να ελέγξουμε το μήκος του μακρύτερου μονοπατιού που αντιστοιχεί σε κάποιον st-προσανατολισμό. Η σημαντικότητα αυτής της ερευνητικής κατεύθυνσης είχε τονισθεί στο παρελθόν, ειδικότερα στο χώρο της σχεδίασης γράφων.

Οι αλγόριθμοι που παρουσιάζονται υπολογίζουν st -προσανατολισμένους γράφους με μήκος μακρύτερου μονοπατιού που μπορεί να καθοριστεί απο παραμέτρους. Το μήκος αυτό είναι πολύ σημαντικό στην ποιότητα των λύσεων που παράγονται από διάφορους αλγορίθμους. Για παράδειγμα, τα όρια εμβαδού πολλών αλγορίθμων σχεδίασης γράφων εξαρτώνται από το μήκος του μακρύτερου μονοπατιού του st-προσανατολισμένου γράφου. Επιπλέον, συγκεκριμένοι st-προσανατολισμοί γράφων μπορούν να προσεγγίσουν καταλλήλως μορφοποιημένα προβλήματα

γράφων (μακρύτερο μονοπάτι, χρωματισμός γράφων). Τέλος, η δρομολόγηση δικτύων μέσω st-αριθμήσεων δίνει πολλάπλα μονοπάτια προς κάποιο κόμβο και συνεπώς ο υπολογισμός διαφορετικών (παραμετρικών ως προς το μακρύτερο μονοπάτι) st-αριθμήσεων παρέχει ελαστικότητα σε πολλά υπάρχοντα πρωτόκολλα δρομολόγησης δικτύων. Εξετάζουμε τις περισσότερες από αυτές τις εφαρμογές και δείχνουμε ότι υπάρχει πραγματικά η ανάγκη συστηματικού υπολογισμού παραμετρικών st-προσανατολισμών.

Acknowledgements

First of all, I would like to thank my supervisor Professor Ioannis G. Tollis for his excellent scientific guidance, his introducing me into the problem, his interest in producing original work and for showing me the ”difficult” way from the very first day we met. I also thank him for the so many afternoon meetings that always put me into creative thought for many weeks.

I would also like to thank Professor George F. Georgakopoulos for his useful comments on my work and Professor Evangelos Markatos for his comments on the final version of the thesis.

Special thanks also deserve to Professors H. de Fraysseix (CNRS), M. Yannakakis (Columbia University), C. Papadimitriou (University of California, Berkeley), R. Tarjan (Princeton University) with whom I developed a very fruitful conversation through e-mails about various aspects of my work. Additionally, I could not forget late Professor Shimon Even's (Israel Institute of Technology) contribution to my work, as he had the time to mail me the original versions of two of his papers back in November 2003. I would also like to thank Dr. Martin Doerr, with whom I had another very fruitful collaboration at the Information Systems Laboratory of the Institute of Computer Science of the Foundation for Research and Technology Hellas (ICS-FORTH), far from this thesis. Finally, I thank my undergraduate advisors Professors K. Paparrizos and N. Samaras (University of Macedonia) for inspiring me to work on the exciting field of Algorithms.

I have also the need to express my thanks to the Institute of Computer Science of the Foundation for Research and Technology Hellas (ICS-FORTH) and the University of Crete for financially supporting me during the last two years in Crete.

Above all, I owe a lot to my friends (new and old) for always being there for me in difficult and nice moments. Last, but not least, I thank my family for their carry and support in all aspects of my life.

Heraklion, Crete — Charalampos Papamanthou
July 2005

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 [22, 18, 3, 4, 21], network routing [2, 1] and graph partitioning [17]. Most algorithms use any algorithm that produces such an orientation, e.g., [8], without expecting any specific properties of the oriented graph. In this thesis we present new algorithms 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 [22, 18, 2, 1, 17].

Given a biconnected undirected graph $G = (V, E)$, with n vertices and m edges, and two nodes s and t, an st-orientation (also known as bipolar orientation or st-numbering) 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. An st-orientation of an undirected graph can easily be computed using an st-numbering [8] of the respective graph G and orienting the edges of G from *low* to *high*. An st-numbering 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.

st-numberings were first introduced in 1967 in [15], 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 [15] 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 [8]. Ebert [7] presented a slightly simpler algorithm for the computation of such a numbering, which was further simplified by Tarjan [24]. The planar case has been extensively investigated in [19] where a linear time algorithm is presented which may reach any st-orientation of a planar graph. Finally, in [16] a parallel algorithm is described. An overview of the work concerning bipolar orientations is presented in [9].

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 [8, 7, 24]. 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 [18, 19]. Our algorithms are able to compute *st*-oriented graphs of absolutely ”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 [22, 18, 21] 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 [2, 1].

The thesis is organized as follows: Chapter 1 presents some results and algorithms from the past and points out the need for an algorithm that computes *longest path* parameterized *st*-orientations. Chapter 2 presents a new algorithm for computing an *st*-orientation of a general undirected biconnected graph. Chapter 3 presents some techniques that can be implemented on the developed algorithm in order to control the length of the longest path of the final *st*-oriented graph. Chapter 4 discusses applications of parameterized *st*-orientations (primal and dual parameterized *st*-orientations, graph coloring, longest path) and *st*-orientations of special classes of graphs. Chapter 5 presents the overall computational results. Finally, Chapter 6 talks about open problems and future work.

Chapter 1

Background Work

1.1 Basic Definitions

We begin with two very important definitions, the st-numbering and st-orientation, and discuss how these definitions are mutually dependent:

Definition 1.1.1. *Let $G = (V, E)$ be an undirected biconnected graph. Let (s, t) be one of its edges. An st-numbering is a function $f : V \to \{1, \ldots, n\}$ such that $f(s) = 1$, $f(t) = n$ and $\forall v \in V - \{s, t\}$ there are two edges (x, v) and (v, y) such that $f(x) < f(v) < f(y)$.*

Definition 1.1.2. *Let $G = (V, E)$ be a directed graph. G is st-oriented if and only if it has one single source s, one single sink t and contains no cycles.*

It is easy to prove that G has an st-orientation if and only if it has an st-numbering and we can compute either from the other in $O(n+m)$ time, as follows. Given an st-orientation, we number the vertices of G in topological order using Knuth's algorithm [14]. This produces an st-numbering. Given an st-numbering, we orient each edge from its lower-numbered to its higher-numbered endpoint. This produces an st-orientation.

Note that computing an st-orientation from an st-numbering is a 1-1 function. On the opposite, there may exist more than one st-numberings that correspond to a certain st-orientation. Finally, if G is st-oriented, for each node $v \in V$ there exists a directed path from s to t that contains v. For a complete review of the properties of st-oriented planar graphs (st-planar graphs), see [4].

In 1967, Lempel, Even and Cederbaum [15] made a first approach to this problem, by presenting an $O(nm)$ time algorithm for the computation of an st-numbering of the vertices of an undirected graph in order to check whether a graph is planar or not. They proved the following result:

Theorem 1.1.3 (Lempel et all [15]). *Let (s, t) be any edge of a graph G. Then, G admits an st-orientation if and only if G is biconnected.*

Proof. Assume G admits an *st*-orientation. Let $v \neq \{s,t\}$ be a vertex of G. We will prove that $G - \{v\}$ is still connected. Let $x \neq \{s,t,v\}$ be a vertex of G. As G is *st*-oriented there will always be a directed path from s to t that contains x. The vertex x is connected to s or t in $G - \{v\}$. As s,t are adjacent, $G - \{v\}$ is connected. Similarly, if we delete the vertex s or t, each vertex x is connected to t or s. Thus G is biconnected. Finally, let (s,t) be an edge of a biconnected graph and let γ be a cycle that contains (s,t). Fraysseix et all [9] have proved that an *st*-orientation of every partial subgraph H of a biconnected graph G can be extended to an *st*-orientation of G. Hence the *st*-orientation of γ can be extended to an *st*-orientation of G. In [15], this proof is given with an $O(nm)$ algorithm that *st*-orients every biconnected graph G. □

In the following years, more efficient algorithms for the computation of such a numbering were devised. Actually, these algorithms are based on the well known *depth first* search traversal [23] (DFS) of graphs and run in linear time $O(n+m)$. Following we present two of the most important and widely used algorithms.

1.2 The Even-Tarjan Algorithm

In 1974, Even and Tarzan [8] developed an $O(n+m)$ algorithm for the computation of an *st*-numbering. The algorithm is based on DFS and uses the circles formed during the execution of a DFS. As it is already known, given an undirected connected graph $G = (V, E)$ we can execute a DFS and get a DFS tree. All nodes v of the initial graph are contained in the tree and get a number $d(v)$ which actually denotes the rank of their visit.

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

$$u - ... - v \Leftrightarrow u \hookrightarrow v \mid v \hookrightarrow u$$

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

$$L(x) = \min(\{d(x)\} \cup \{d(y) : \exists w : x \hookrightarrow w \wedge w - ... - y\}) \tag{1.2.1}$$

Note that the lowpoint function is not an 1-1 function, i.e. there can be two nodes getting the same lowpoint. It is easy to see from 1.2.1 that a node x either gets its DFS number as a lowpoint or the DFS number of a node y, previously visited by DFS, reachable from x by following a downward tree path to a node w which ends with a cycle edge from w to y. This path may contain no tree edges. Next, we will present a lemma that comes out of the definition of the lowpoint function.

Lemma 1.2.1 (Tarjan [23]). *If G is biconnected and $v \to w$, then $d(v) \neq 1$ implies $L(w) < d(v)$ and $d(v) = 1$ implies $L(w) = d(v) = 1$.*

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

The values $L(v)$ can easily be computed in time $O(n+m)$ during the execution of DFS. We will now describe the algorithm for the computation of an *st*-numbering. We are given an undirected biconnected graph $G = (V, E)$ and we want to assign numbers to its vertices which satisfy the definition of *st*-numbering. Let (s, t) be one edge of G. In the beginning, we execute a DFS, such that the root of the DFS tree is node t and the first edge of the tree is $t \to s$. During DFS, we also compute the lowpoint numbers $L(v)$ for every node v. The information generated by DFS is valuable for the remaining part of the algorithm.

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

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

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

Algorithm 1 PATHFINDER(v)

```
if ∃v − ... − w ∈ U_c new with w ↪ v then
    mark (v, w) as old;
    p = {v, w};
else if ∃v → w ∈ U_t new then
    mark (v, w) as old;
    p = {v, w};
    while w new do
        find new (w, x) with (d(x) = L(w) | (L(x) = L(w) ∧ w → x));
        mark w and (w, x) as old;
        p = p ∪ (w, x);
        w = x;
    end while
else if ∃v − ... − w ∈ U_c new with v ↪ w then
    mark (v, w) as old;
    p = {v, w};
    while w new do
        find new (w, x) with x → w;
        mark w and (w, x) as old;
        p = p ∪ (w, x);
        w = x;
    end while
else
    p = {∅};
end if
return p;
```

Lemma 1.2.2. *Supppose vertices s, t and edge (s, t) are initially marked old. An initial call $PATHFINDER(s)$ will return a simple path from s to t not containing (s, t). A successive call $PATHFINDER(v)$ with v old will return a simple path (of edges new before the call) from v to some vertex w old before the call, if there are any edges (v, w) new before the call (otherwise PATHFINDER(v) returns the null path).*

Proof. It easy to prove by induction on the number of the $PATHFINDER$ calls that, at the beginning of any $PATHFINDER$ call, if some vertex w is old, then all vertices and edges on the tree path from t to w are old. Given this fact, we can prove the Lemma by considering the four different choices made at lines 1,4,13,22 of the $PATHFINDER$ procedure. If choices at line 1 or 22 are made, $PATHFINDER$ obviously performs according to the statement of the Lemma. Consider the choice made at line 4. By Lemma 1.2.1, $L(w) < d(v)$, where (v, w) is the first edge on the path. Thus statement at line 4 selects some path $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$, where $v_i \to v_{i+1}$ for $1 \leq i < k$, $v_{k-1} - ... - v_k$, $v_k \hookrightarrow v_{k-1}$ and $d(v_k) = L(v_2) < d(v_1)$. Hence the selected path is simple. Consider choice at line 13. Since choice at line 4 is not made, all vertices x such that $v \to x$ are old when choice at line 13 is made. Thus the selected path terminates at some descendant of v (not v) and is simple. □

The main algorithm for the computation of an *st*-numbering uses the pathfinder procedure to compute an *st*-numbering. The pseudocode of the algorithm is given (see Algorithm 2). In the following, we will prove the correctness of the algorithm. The importance and

Algorithm 2 STNUMBER(G, s, t)

```
compute the lowpoints L(v) for all nodes v ∈ V;
mark s, t and (s,t) as old and all other vertices and edges as new;
initialize a stack R;
R =push(t);
R =push(s);
i = 0;
while R ≠ Ø do
  v=pop(R)
  p = {{v_1, v_2}, {v_2, v_3}, ..., {v_{k-1}, v_k}} = PATHFINDER(v);
  if p ≠ Ø then
    for j = k − 1 downto 1 do
      R =push(v_j)
    end for
  else
    i = i + 1;
    f(v) = i;
  end if
end while
```

efficiency of the algorithm depends on the clever use of the stack.

Theorem 1.2.3. *Algorithm $STNUMBER$ correctly computes an st-numbering of an undirected biconnected graph $G = (V, E)$.*

Proof. It is evident that no vertex v appears in two or more places on stack at the same time. Once a vertex v is placed on stack, nothing under v receives a number until v does.

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

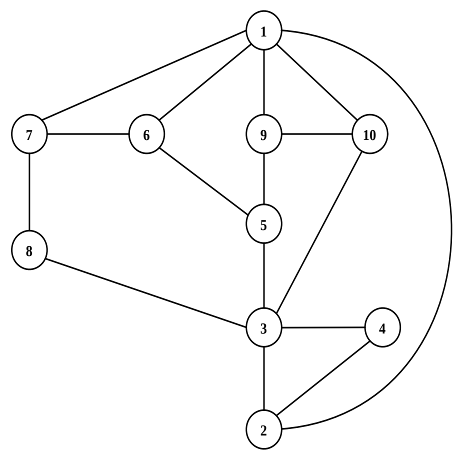


Figure 1.1: A biconnected graph G.

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

Let as now regard an undirected graph G of 10 vertices (see Figure 1.1). Graph G is biconnected and thus we can apply our algorithm to find an st-numbering. We will find a 2-1-numbering. The reader can verify that during the algorithm execution the variables of

Table 1.1 will be computed. The final vector produced by the algorithm is

$$f = \begin{bmatrix} 10 & 1 & 3 & 2 & 7 & 6 & 5 & 4 & 8 & 9 \end{bmatrix}$$

Note that $f(2) = 1$ and $f(1) = 10$. Additionally, vector f satisfies the st-numbering definition.

Table 1.1: The Even-Tarjan algorithm execution.

iteration #	**stack status**	**path**	**operation**
1	{1,2}	$2 \rightarrow 3 \rightarrow 8 \rightarrow 7 \rightarrow 1$	
2	{1,7,8,3,2}	$2 \rightarrow 4 \rightarrow 3$	
3	{1,7,8,3,4,2}	**null**	$f(2) = 1$
4	{1,7,8,3,4}	**null**	$f(4) = 2$
5	{1,7,8,3}	**null**	$f(3) = 3$
6	{1,7,8}	**null**	$f(8) = 4$
7	{1,7}	$7 \rightarrow 6 \rightarrow 1$	
8	{1,6,7}	**null**	$f(7) = 5$
9	{1,6}	$6 \rightarrow 5 \rightarrow 9 \rightarrow 1$	
10	{1,9,5,6}	**null**	$f(6) = 6$
...	...	...	...
15	{1,10}	$10 \rightarrow 3$	
16	{1,10}	**null**	$f(10) = 9$
17	{1}	**null**	$f(1) = 10$

1.3 A Streamlined Depth First Search Algorithm

Another simpler algorithm for the computation of an st-numbering was proposed by Tarjan in 1986 [24]. The algorithm is also based on a DFS traversal of the initial biconnected graph. In the depth first tree, we denote with $p(v)$ the father of node v. The algorithm works as follows.

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

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

if $s(L(v)) == *+$ **then**
  Insert v after $p(v)$ in $\wp$;
  $s(p(v)) = *-$;
end if
if $s(L(v)) == *-$ **then**
  Insert v before $p(v)$ in $\wp$;
  $s(p(v)) = *+$;
end if

Theorem 1.3.1. *The st-numbering is correct.*

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

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

For the first case, suppose $s = x_0$, $t = x_1, x_2, \ldots, x_l$ be the tree path from s to the vertex x_l most recently added to $\wp$ and let v with parent x_k be the next vertex to be added to $\wp$. Assume as an induction hypothesis that for all $0 \leq i < j < l$, $s(x_i) = *+$ if and only if x_i follows x_j in $\wp$, i.e., $x_i = p(x_j)$. Since $s(x_k)$ is set to *minus* if v is inserted after x_k in $\wp$ and to *plus* if v is inserted before x_k in $\wp$, the induction hypothesis holds after v is added. Hence the induction holds.

For the second case, let $v \neq s, t$. If $(v, L(v))$ is a back edge, the insertion of v between $p(v)$ and $L(v)$ in $\wp$ guarantees that in the numbering corresponding to $\wp$, v is adjacent to both a lower-numbered and a higher-numbered vertex. Otherwise, there must be a vertex w such that $p(w) = v$ and $L(w) = L(v)$. By Lemma 1.2.1 we have that $L(v)$ is a proper ancestor of v, which means that $s(L(v))$ remains constant during the time v and w are added to $\wp$. It follows that v appears between $p(v)$ and w in the completed list $\wp$, which implied that in the numbering corresponding to $\wp$, v is adjacent to both a lower-numbered and higher-numbered vertex. Thus, the second case holds. $\square$

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

Following, we give an execution example of the algorithm. Suppose we want to compute a 2-1-numbering of the biconnected graph of figure 1.1 using the streamlined DFS algorithm. First we execute a DFS, and we compute the DFS tree and the lowpoint values. The lowpoint values of each vertex in Figure 1.2 are depicted with bold numbers. In Table 1.2, we can see the variables computed by the algorithm. Before the development of the streamlined DFS algorithm by Tarjan, Ebert [7] presented a more complicated algorithm for the computation of such a numbering, on which the Tarjan's algorithm was based.

Additionally, the planar case has been extensively investigated in [19] where a linear time algorithm is presented which may reach any st-orientation of a planar graph. Finally, in [16]

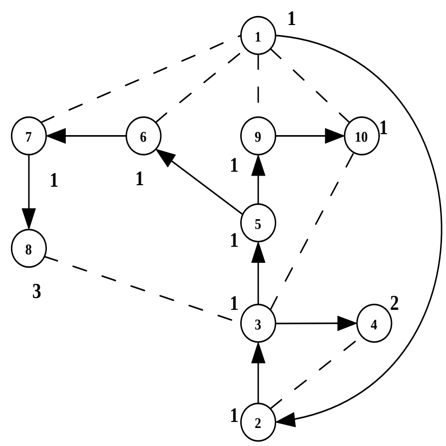


Figure 1.2: The DFS tree of the graph of Figure 1.1.

Table 1.2: The algorithm execution. Irrelevant signs are omitted

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

a parallel algorithm is described. The last solution to the problem was given by Brandes [5], where an algorithm (that does not need lowpoint values) for computing an st-numbering is presented. An overview of the work concerning bipolar orientations is presented in [9].

1.4 The Need for Parameterized st-Orientations

It is obvious that a biconnected graph G can be st-oriented in multiple ways and st-numbered in many more, as there is a one to many correspondence between st-orientations and st-numberings. In general there is an exponential number of st-orientations that correspond to a certain biconnected graph. Actually, this number is connected to the chromatic polynomial of a graph G, as proved by Stanley in 1973 [20].

st-orientations and st-numberings, as mentioned in the introduction, are very important to many applications. They are used by many algorithms in their first step. Therefore it would be desirable to try define a metric that corresponds to an st-orientation so that one could choose between different st-orientations. One of the most important variables that characterize an st-oriented graph is the **length** of the longest path from s to t, denoted with $l(t)$, and which can be computed in $O(n+m)$ time, given an st-orientation.

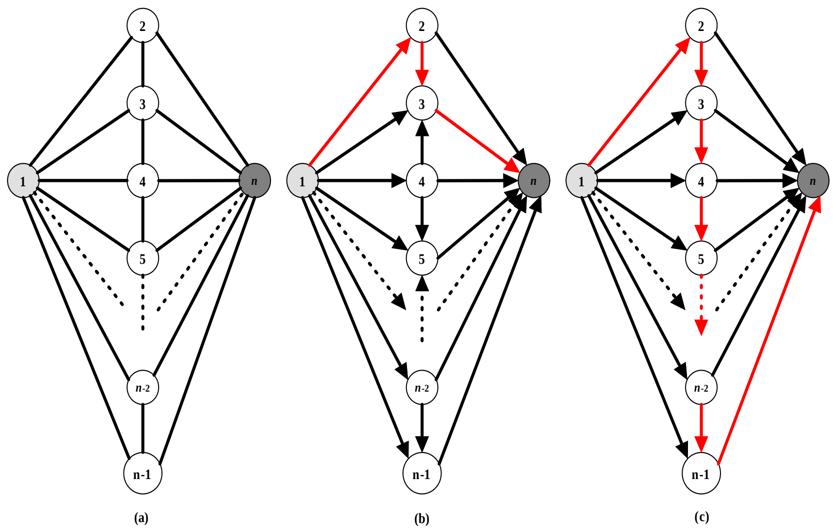


Figure 1.3: An undirected biconnected graph (a) and two different st-orientations of it ((b),(c)) (of different length of longest path from s to t).

Suppose we are given the undirected biconnected graph G of Figure 1.3. If we apply the existing algorithms we compute a random st-orientation of random longest path length $l(t)$. Note that G has exactly $n-3$ st-orientations of longest path length $l(t) = 3, 4, \ldots, n-1$. The question that arises is evident: Can we devise an algorithm that computes st-orientations of *almost* predefined longest path length $l(t)$? If we could have such an algorithm, we could guarantee the computation of different st-orientations, something that will give as the opportunity to choose an *ideal* st-orientation, according to the application. For example, in Graph Drawing, visibility representations algorithms such as [22] or hierarchical drawing algorithms such as [21], produce drawings that are strongly dependent on the length of the longest path of the st-orientation.

The contribution of this thesis is the development of a new algorithm that simultaneously computes both an st-orientation and an st-numbering of a biconnected graph G and also uses input parameters to control the length of the longest path of the final st-oriented graph, $l(t)$. The importance of this research direction has been implied in the past [18, 19].

Chapter 2

A New Algorithm for Computing an st-Orientation

2.1 Introduction

In this chapter, we present the newly developed algorithm that computes an st-orientation of a biconnected graph $G = (V, E)$. We analyze its behavior and give proof of correctness. This algorithm is designed in such a way that gives us the opportunity to develop techniques that actually *define* the length of the longest path of the final st-oriented graph. For the rest of the thesis, $n = |V|$, $m = |E|$, $N_G(v)$ 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 the presentation of the algorithm by presenting its function 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 removal of vertices (for example the K_n graphs).

Definition 2.2.1. *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 $G_i = G_{i-1} - \{v_{i-1}\}$, $v_i \in N_{G_{i-1}}(v_{i-1}) \sim \{t\}$, $i = 2, \ldots, n-1$ and $G_1 = G$ are biconnected.*

Following we present a Lemma that gives an algorithm for the transformation of an st-recursively biconnected undirected graph to an st-oriented graph.

Lemma 2.2.2. *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.

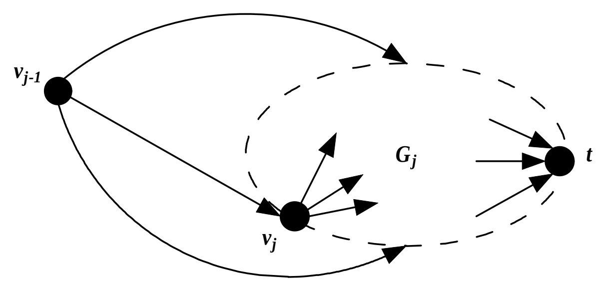


Figure 2.1: Proof of Lemma 2.2.2.

It remains to see that the directed graph produced by following this procedure is st-oriented. Suppose we have removed all nodes and we have computed the edge directions. We backtrack from $v_n = t$ by adding the computed directed edges to a new graph $F = (V', E')$ and finally conclude (by induction) that F is st-oriented: The last directed edge computed is the edge (v_{n-1}, t). Let F_{n-1} be the directed graph that contains (v_{n-1}, t). F_{n-1} is $v_{n-1}t$-oriented and the base case holds. Suppose after the k-th backtracking F_{n-k} is $v_{n-k}t$-oriented. After the $(k+1)$-th backtracking, F_{n-k-1} is $v_{n-k-1}t$-oriented, as a new source v_{n-k-1} is added to an already $v_{n-k}t$-oriented graph F_{n-k}. This source is connected through a directed edge (v_{n-k-1}, v_{n-k}) with the previous source and hence v_{n-k} is no longer a source and no cycles are created. By induction, F_1 is an st-oriented graph and the Lemma holds. □

Following, we give the recursive algorithm (Algorithm 3) for computing an st-orientation of an st-recursively biconnected graph as implied in Lemma 2.2.2. Note that, in order to use this algorithm, we must know the permutation P on which G is st-recursively biconnected. This algorithm has no practical interest but is certainly an introduction to the problem we will tackle next.

Algorithm 3 runs in $O(nm)$ time in the worst case, as the recursion is executed exactly $n-1$ times, the main body (lines 8-12) of the algorithm takes at most $O(m)$ and lines 13-17 (taking into consideration that we know the permutation P on which G is recursively biconnected) take time $O(1)$.

In Figure 2.2a, the execution of Algorithm 3 is depicted. The selected sources are 1,2,3,4. Note that all the graphs produced after the removal of these sources are biconnected. As shown in Figure 2.2a, the length of the longest path of the final produced st-oriented graph is maximum (=4). If sources are chosen in a different way, not according to Lemma 2.2.2, we can obtain an st-oriented graph with lower longest path length. This is shown in Figure 2.2b, where sources are chosen in the following sequence: 1,3,2,4. The removal of vertex 3 results in an one-connected graph (the path $2 \rightarrow 4 \rightarrow 5$) and the longest path length is $3 < 4$.

Algorithm 3 STNRB(G, s, t)

1: Initialize $F = (V', E')$; $\{$*F is the final st-oriented graph*$\}$
2: $Q = \{s\}$; $\{$*Insert s into Q*$\}$
3: STREC(G, s); $\{$*Call the recursive algorithm*$\}$
4: ————————————————————-
5: **function** STREC(G, v)
6: $V = V - \{v\}$; $\{$*A source is removed from G*$\}$
7: $V' = V' \cup \{v\}$; $\{$*and is added to F*$\}$
8: **for all** edges $(v, i) \in E$ **do**
9:     $E = E - \{(v, i)\}$;
10:     $E' = E' \cup \{(v, i)\}$;
11: **end for**
12: $Q = \{i : i \neq t \wedge (v, i) \in E'\}$; $\{$*The set of possible next sources*$\}$
13: **if** $Q == \{\emptyset\}$ **then**
14:     **return**;
15: **else**
16:     choose $u \in Q$ such that $G - \{u\}$ is biconnected;
17:     STREC(G, u);
18: **end if**

Corollary 2.2.3. *Let $G = (V, E)$ be an undirected biconnected graph and s, t two of its nodes. Lemma 2.2.2 can produce up to $(n-2)!$ st-oriented graphs. Moreover, this bound is achieved for the K_n graph.*

Proof. At each stage of the recursive procedure described in Lemma 2.2.2, a source that does not "*disbiconnect*" the graph is removed. If there is always exactly one such source, then we produce exactly one st-oriented graph. In the case of the K_n graph, all nodes, except for t, can be chosen as future sources, as we must remove exactly $n-1$ nodes to loose biconnectivity. This means that when a node v_i is removed, we have exactly $|N_{G_i}(v_i) \sim \{t\}| = n-i-1$ choices to continue. Hence we get exactly $\prod_{i=1}^{n-1}(n-i-1) = (n-2)!$ st-oriented graphs (see Figure 2.3). $\square$

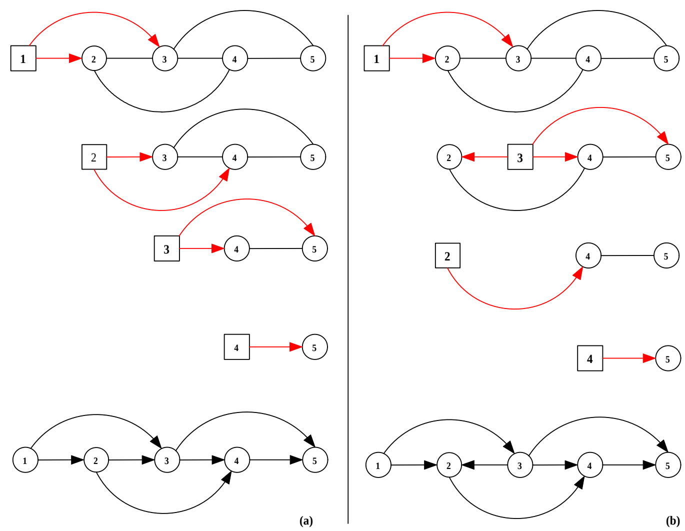


Figure 2.2: (a) The execution of Algorithm 3 (the current source is each time depicted with a rectangle), (b) An alternative choice of sources yields lower longest path length.

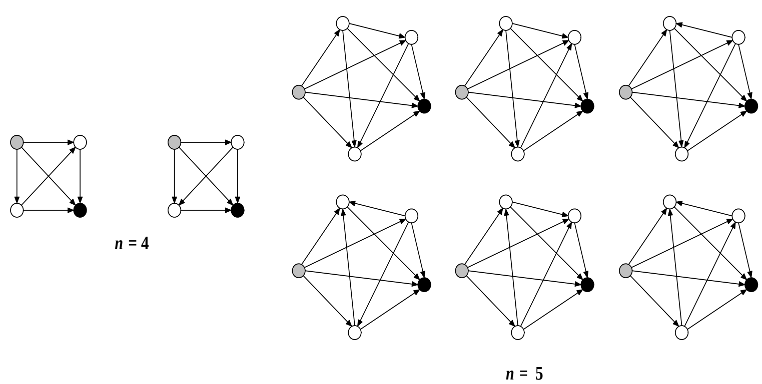


Figure 2.3: K_n st-oriented graphs. Sources are the gray nodes, whereas sinks are the black nodes.

Corollary 2.2.4. *Algorithm 3 produces st-oriented graphs of maximum longest path length*

$l(t) = n - 1$.

Proof. Immediate from Lemma 2.2.2, as the nodes that are gradually removed lie on a common path from s to t and finally all nodes are removed by the algorithm. Hence, the longest path length will be $n - 1$. □

2.3 General Graphs Case

2.3.1 Preliminaries

In the previous section, we examined a special class of graphs. We now present the general case, where there is no other option than to remove a node that produces a one-connected subgraph. Before continuing with this section, we will introduce some useful terminology.

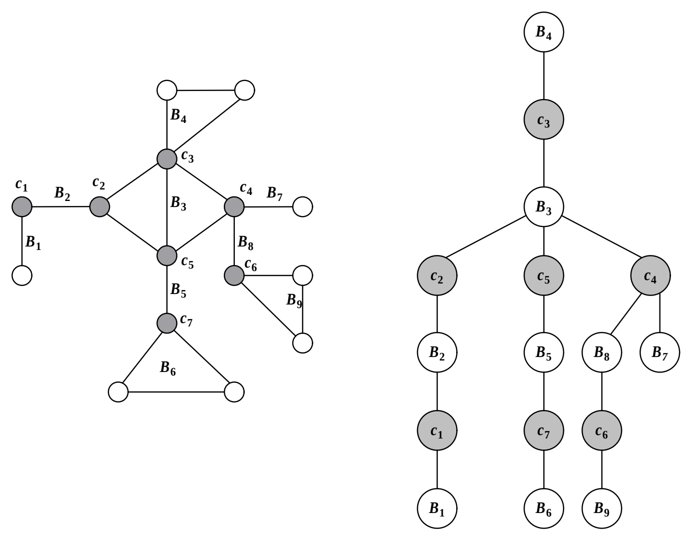


Figure 2.4: A one-connected graph and the t-rooted block-cutpoint tree rooted on B_4

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 Figure 2.4).

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 Figure 2.4).

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 [13]. Following, we give some results that are necessary for the development of the algorithm.

Lemma 2.3.1. *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.

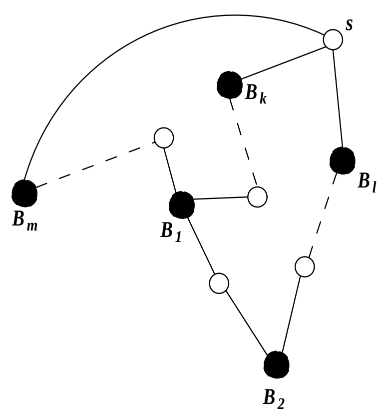


Figure 2.5: Proof of Lemma 2.3.1.

If graph $G - \{s\}$ is one-connected (see Figure 2.5), 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. $\square$

As each t-rooted block-cutpoint tree will have at least one leaf-block, we have:

Corollary 2.3.2. *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 a leaf-block of the t-rooted block-cutpoint tree of $G - \{s\}$. Moreover, this neighbor 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

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.

2.3.2 The Algorithm

Now we describe the procedure 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 procedure 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. 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 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 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 2.3.3. *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 Corollary 2.3.2, 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 combining Lemmas 2.3.1, 2.3.3 and Corollary 2.3.2, we see that at each iteration of the algorithm there will be at least one node to be chosen as a future source:

Corollary 2.3.4. *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 2.3.5. *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 2.3.6. *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. $\square$

Algorithm 4 STN(G, s, t) (rec)

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_ℓ);
    end for
end if

By Lemmas 2.3.5, 2.3.6 we have:

Theorem 2.3.7. *The directed graph $F = (V', E')$ is st-oriented.*

Algorithm 4 is the STN pseudocode 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 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 $O(n+m)$ [13]. 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 [12] 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 2.3.8 (Holm, Lichtenberg and Thorup [12]). *There exists a deterministic fully dynamic algorithm for maintaining biconnectivity in a graph, using $O(\log^5 n)$ amortized time per operation (edge insertion or deletion).*

Therefore, if we use the above algorithm for the biconnectivity maintenance, the time complexity of our algorithm can be clearly reduced to $O(m \log^5 n)$. Hence we have the following:

Theorem 2.3.9. *Algorithm STN can be implemented to run in $O(m \log^5 n)$ time.*

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. As it

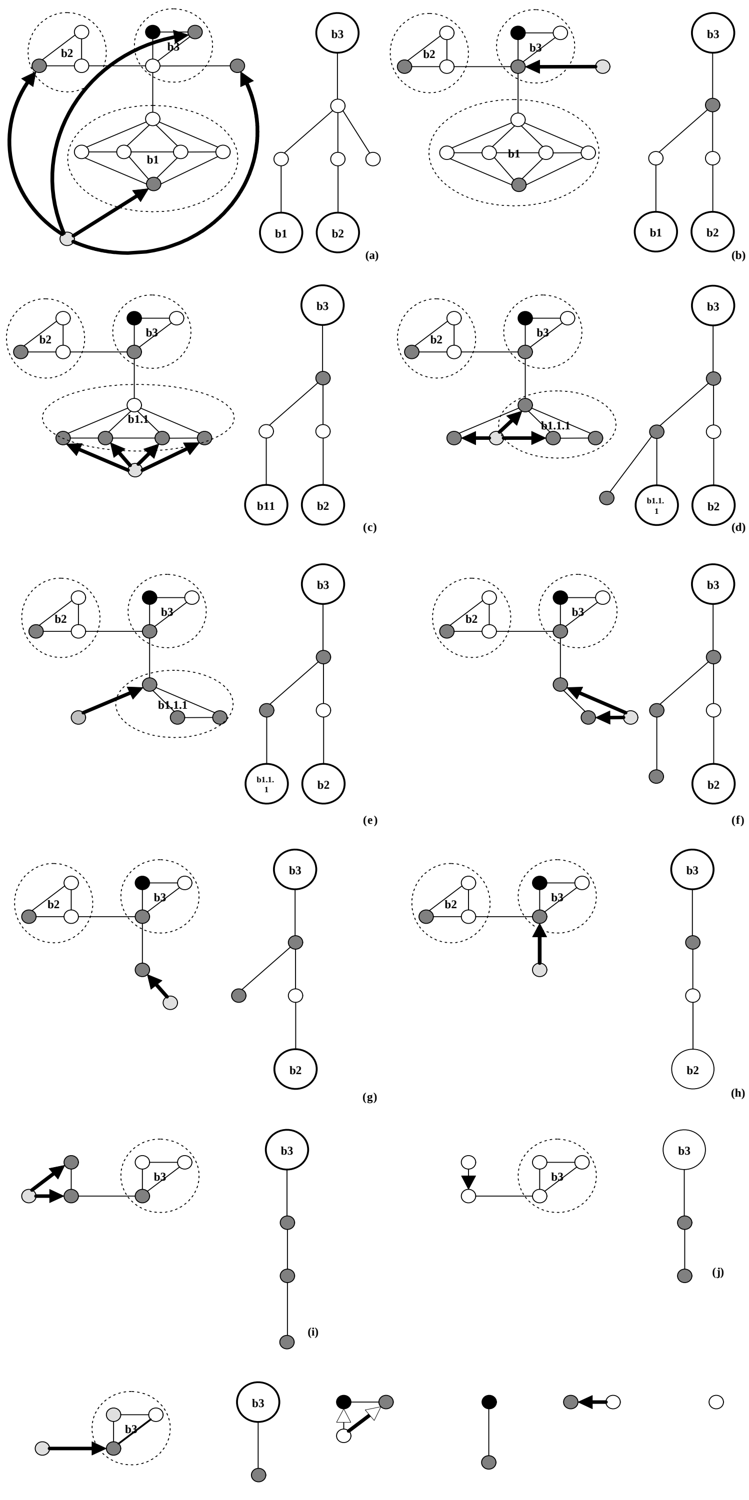


Figure 2.6: The algorithm execution.

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 2.3.10. *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 Figure 2.6, the algorithm execution on a biconnected graph G is depicted. In Figure 2.7, we can see the final st-oriented graph F and the respective st tree T_s. Algorithm 4 can also be implemented non-recursively. Actually, for large-size graphs, we can only use the following non-recursive algorithm (Algorithm 5) in order to avoid stack overflow problems.

Algorithm 5 STN(G, s, t) (non-rec)

1: $Q = \{s\}$; $\{$*insert s into Q*$\}$
2: $j = 0$;$\{$*Initialize a counter*$\}$
3: Initialize $F = (V', E')$;
4: Initialize the t-rooted block-cutpoint tree T to be graph G; Its cutpoint is sink t;
5: **while** $Q \neq \emptyset$ **do**
6:    **for** all leaf-blocks B_j^i **do**
7:       $j = j + 1$;
8:       ***choose*** $v_\ell \in B_j^\ell \cap Q \sim \{h_j^\ell\}$; $\{h_j^\ell$ *is the cutpoint that defines* $B_j^\ell\}$
9:       $f(v_\ell) = j$;
10:       $V = V - \{v_\ell\}$ $\{$*a source is removed from G*$\}$
11:       $V' = V' \cup \{v_\ell\}$ $\{$*and is added to F*$\}$
12:       **for** all edges $(v_\ell, i) \in E$ **do**
13:          $E = E - \{(v_\ell, i)\}$;
14:          $E' = E' \cup \{(v_\ell, i)\}$;
15:       **end for**
16:       $Q = Q \cup \{N(v_\ell) \sim t\} - \{v_\ell\}$; $\{$*the set of possible sources*$\}$
17:    **end for**
18:    $T(t, B_j^1, B_j^2, \ldots, B_j^r)$=**UpdateBlocks**$(G)$;
19: **end while**
20: **return** F, g;

Algorithm 5 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

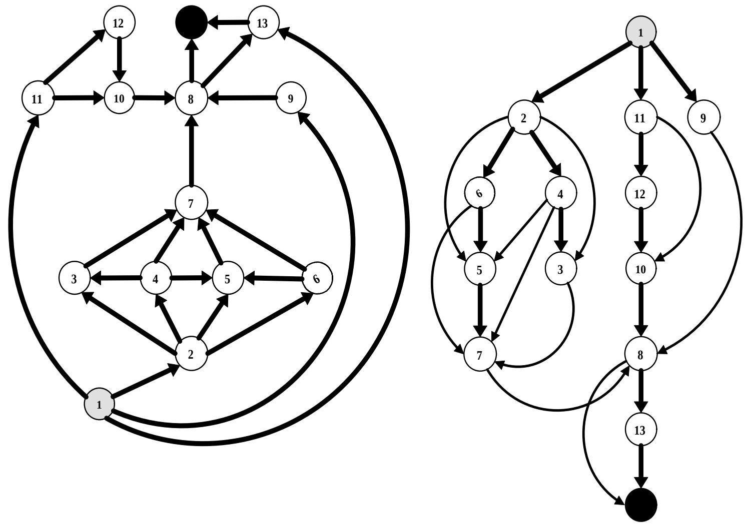


Figure 2.7: The final st-oriented graph (left) and the st-tree T_s (right).

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.

Chapter 3

Longest Path Parameterized st-Orientations

3.1 General

As stated in Chapter 2, our algorithm aims at producing st-oriented graphs of predefined longest path length, i.e., to determine the "quality" of the produced st-oriented graphs. 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 choices the algorithm gives us. Note that in the classical algorithms for st-numbering computation [8], there is no clear way to influence the longest path length.

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 24 of the Algorithm 4):

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

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. As presented in Chapter 2, during each iteration of the algorithm a timer j (line 8 of Algorithm 4) 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. 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. 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 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, we must make a very important remark. During STN, when a node u is removed from the graph, a sink set W is formed. This set contains nodes that were most recently explored. It is computed as follows. Initially, W contains the source of the graph s, i.e., $W = \{s\}$. Let $F = (V', E')$ be the directed graph that is constructed during the execution of the algorithm. For each node v_i, $i = 2, \ldots, n-1$ that is removed from the graph during the iterations of the algorithm the sink set W is updated by inserting node v_i and by removing every node x for which $(x, v_i) \in E'$. Note that (x, v_i) is a directed edge of graph F.

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 end up to 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 to the so far produced sW-oriented graph or apply a relaxation method during the execution of the algorithm (see in next sections), and compute $l(u)$:

Remark 3.1.1. *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-oriented graph is also $l(u)$.*

This remark is very important because it gives us a sense of how the developed algorithm can be related with the length of the longest path.

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 $n - 1$:

Definition 3.1.2. *Let $G = (V, E)$ be an undirected biconnected graph. We say that G is st-Hamiltonian if and only if G has a simple path from a node s to a node t that includes all the other vertices of G.*

3.2 Maximum Case (MAX-STN)

Lemma 3.2.1. *Let $G = (V, E)$ be an undirected st-Hamiltonian graph. MAX-STN 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 MAX-STN 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 formed 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 $l(t) < n-1$, which does not hold and the direct is proved.

The inverse of Lemma 3.2.1 is the following statement: If the block-cutpoint tree always contains one leaf-block, then MAX-STN produces an *st*-oriented graph of maximum longest path length ($=n-1$) when applied to a *st*-Hamiltonian graph. 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 in 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 $l(t) = n-1$. □

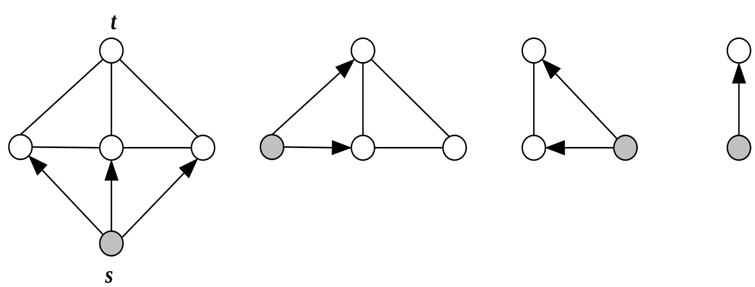


Figure 3.1: Choosing vertices with MIN-STN for a biconnected component that remains biconnected throughout the execution of the algorithm.

Note that the inverse holds only for the case of the MAX-STN procedure. Figure 3.1 provides a counter example showing that if the general STN procedure is applied, a Hamilton 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 achieving this is an NP-hard problem.

MAX-STN 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 chapter. If MAX-STN 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 length. Instead, MAX-

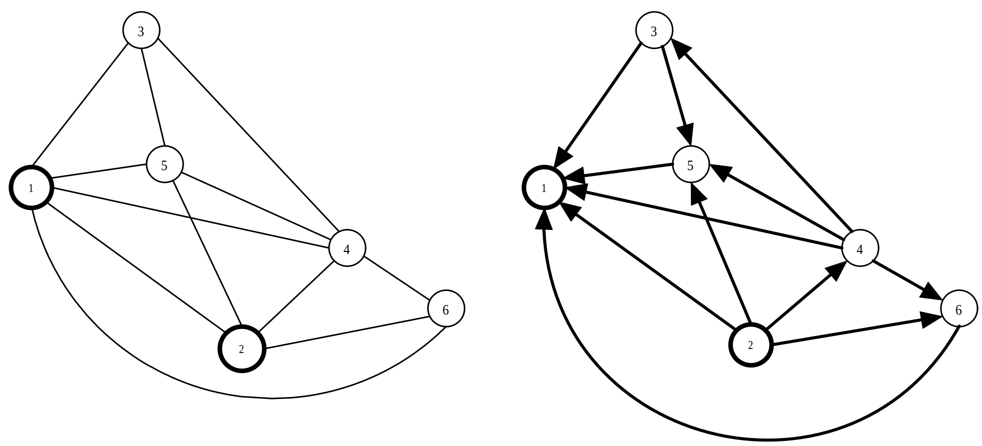


Figure 3.2: MAX-STN applied to a 2-1 Hamiltonian graph. No optimal DAG is produced (longest path length = 4).

STN "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 $k-1$ vertices of the graph have been removed with the following order: $v_1, v_2, \ldots, v_{k-1}$. All r individual paths $P_1, P_2, P_3, \ldots, P_r$ 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 MAX-STN *backtracks*. The length of the longest path of the final *st*-oriented graph is strongly dependent on the number of times that MAX-STN backtracks. All these observations lead to the following remark:

Remark 3.2.2. *Suppose MAX-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 that 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 after the algorithm execution or during the algorithm execution (by applying some a relaxation method).

Figure 3.2 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.

Generally, the length of the longest path computed by the STN algortihm 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 leaf-blocks that are produced during the execution of the algorithm.

Theorem 3.2.3. *Suppose MAX-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 $l(t) \leq n - 1 - \sum_{k_i > k_{i-1}}(k_i - k_{i-1})$.*

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., $l(t) \leq n - 1 - \sum_{k_i > k_{i-1}}(k_i - k_{i-1})$. □

In the experiments conducted on *st*-Hamiltonian graphs we have observed that the length of the longest path computed by MAX-STN is usually very close to $n-1-\sum_{k_i > k_{i-1}}(k_i - k_{i-1})$. Note that Theorem 3.2.3 also holds for the MIN-STN and generally for the STN algorithm.

3.3 Minimum Case (MIN-STN)

MIN-STN 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.

MIN-STN works exactly the same with MAX-STN with the difference that it *backtracks* for a different reason. As we saw before, MAX-STN 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. During this procedure, r different directed paths of vertices are created and the length of the longest path of the final directed graph is always longer than the length of these paths. In MAX-STN, the criterion of backtracking is: *If you encounter a cutpoint, continue execution from the node with the maximum timestamp.* On the other hand, MIN-STN works as follows: It

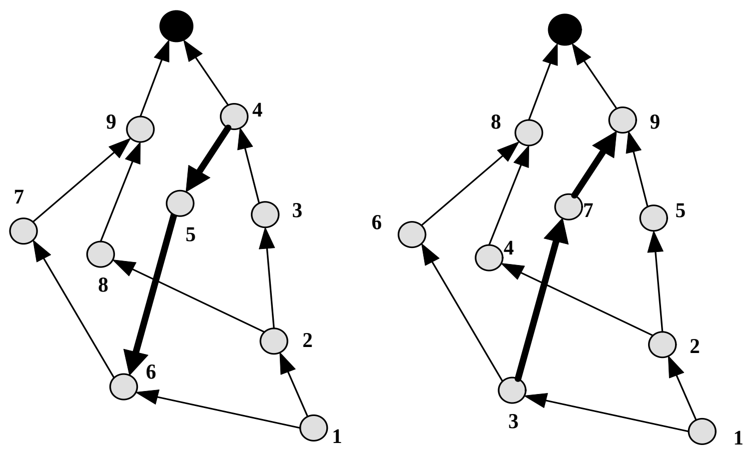


Figure 3.3: MAX-STN (left) and MIN-STN (right) applied to the same biconnected component. The black node is a cutpoint. The thick lines show the different orientation that results in different length of the longest path. The number besides the node represents the visit rank of each procedure.

creates small paths of vertices because backtracking occurs more often, as nodes of minimum timestamp usually lie on previously explored paths. Actually suppose during the execution of MIN-STN r' such paths of vertices $P_1, P_2, P_3, \ldots, P_{r'}$ are created. These paths can be computed with exactly the same algorithm that computes the MAX-STN paths, with the difference that the case $m(v_j) < m(v_{j-1}) + 1$ is likely to occur more times during MIN-STN than during MAX-STN.

3.4 Useful Observations

3.4.1 Longest Path Computations

In this section, we make some observations about both MAX-STN and MIN-STN concerning longest path computations. 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's 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 3.2.2).

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: 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 $u \in R'$ 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 that had previously removed from the graph. Suppose now we enter the k-th iteration of the algorithm and v_k is removed. Let

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

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

Lemma 3.4.1. *Suppose STN enters iteration k and v_k is removed. Then $M_j \leq M_{j-1} + 1$ for all $j = 2, \ldots, k$.*

Actually, Lemma 3.4.1 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.

Suppose now $\ell(v), \lambda(v)$ denotes the length of the longest path from s to a node v computed by MAX-STN and MIN-STN respectively. We analogously define L_k and Λ_k as follows:

$$L_k = \max\{\ell(v_j) : j = 1, \ldots, k\}$$

and

$$\Lambda_k = \max\{\lambda(v_j) : j = 1, \ldots, k\}$$

Conjecture 3.4.2. *Let G be an n-node undirected biconnected graph and s, t be two nodes of its vertex set. Then if we apply MAX-STN and MIN-STN on it, it is $L_n \geq \Lambda_n$.*

If this conjecture finally holds, then $\ell(t) = L_n \geq \lambda(t) = \Lambda_n$ and MAX-STN computes st-oriented graphs of greater or equal longest path length than MIN-STN does. This is actually something that experimentally (as we will see in the Experimental Results section) holds for sure.

To face this conjecture, suppose both MIN-STN and MAX-STN are executed on the same graph G and they enter iteration k. MIN-STN has removed the nodes $v_1, v_2, \ldots, v_k$ and MAX-STN has removed the nodes $w_1, w_2, \ldots, w_k$. We use induction and we describe the problem with the proof. For the base case, it obviously holds $L_1 \geq \Lambda_1$. Suppose, after iteration k it holds $L_k \geq \Lambda_k$. We would like to prove that, concerning the way MAX-STN and MIN-STN work, it is

$$L_{k+1} \geq \Lambda_{k+1}$$

There are four cases:

- $\ell(v_{k+1}) > L_k \wedge \lambda(w_{k+1}) > \Lambda_k$
- $\ell(v_{k+1}) > L_k \wedge \lambda(w_{k+1}) \leq \Lambda_k$
- $\ell(v_{k+1}) \leq L_k \wedge \lambda(w_{k+1}) \leq \Lambda_k$
- $\ell(v_{k+1}) \leq L_k \wedge \lambda(w_{k+1}) > \Lambda_k$

Note that by Lemma 3.4.1, the first three cases imply that $L_{k+1} \geq \Lambda_{k+1}$, given that $L_k \geq \Lambda_k$. The problem is with case 4, where $L_{k+1} = L_k$ and $\Lambda_{k+1} = \Lambda_k + 1$. In this case we would like to prove that

$$L_{k+1} \geq \Lambda_{k+1} \Rightarrow L_k \geq \Lambda_k + 1 \Rightarrow L_k > \Lambda_k$$

If we suppose that $L_k = \Lambda_k$, then we should try to derive something false, for example that the condition $\ell(v_{k+1}) \leq L_k \wedge \lambda(w_{k+1}) > \Lambda_k$ does not hold. The problem with the proof is that in the general case the sequences of vertices $v_1, v_2, \ldots, v_k$ and $w_1, w_2, \ldots, w_k$ are different.

3.4.2 Longest Path Timestamps

Till 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 (by 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) ∈ 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.

3.5 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 MAX-STN (see Lemma 3.2.1) to an st-Hamiltonian graph and find its longest path, which is a well known NP-hard problem [10]. Following we define two decision problems and prove their NP-hardness.

Definition 3.5.1 (Maximum st-Oriented Graph Problem). *Given an undirected biconnencted 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 3.5.2. *The Maximum st-Oriented Graph Problem is NP-hard.*

Proof. We reduce the st-Directed Hamilton Path, which is NP-complete [10], 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

[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.

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 3.5.3 (Minimum st-Oriented Graph Problem). *Given an undirected biconnencted 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?*

Theorem 3.5.4. *The Minimum st-Oriented Graph Problem is NP-hard.*

Proof. We will reduce Graph Coloring to it. Let $G = (V, E)$ be a graph. Suppose we produce $G' = (V', E')$ as follows

$$V' = V \cup \{s, t\}$$

and

$$E' = E \cup \{(s, i)\} \cup \{(i, t)\} \ \forall i \in V$$

We will prove that G can be colored with c colors if and only if the edges of G' can be oriented in a way that the longest path length from s to t is $c + 1$. Then if $\chi(G)$ is the chromatic number of G, the minimum st-orientation of G' will have longest path from s to t equal to $\chi(G) + 1$.

For the direct, suppose we have a coloring of G consisting of c colors. We can orient the edges of G' from the lowest to the highest color. For every st-path p of G', the colors increase along p and hence p has length at most $c + 1$. For the inverse, if we have an st-orientation of G' with L being the longest path from s to t, assign a color to each node u of G equal to the longest path length from s to u. Then we need at most $L - 1$ colors to color the graph. □

3.6 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 MAX-STN for the first pn iterations and the procedure 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 $\Delta(p)$.

Algorithm 6 PAR-STN(G, s, t, p) (rec)

Initialize $F = (V', E')$;
Initialize $m(i) = 0$ for all nodes i of the graph; (timestamp vector)
Initialize $l(i) = 0$ for all nodes i of the graph; (longest path length vector)
$j = 0$; $\{$*Initialize a counter*$\}$
$Q = \{s\}$; $\{$*Insert s into Q*$\}$
PAR-STREC(G, s); $\{$*Call the recursive algorithm*$\}$

---

function PAR-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**
    $E = E - \{(v, i)\}$;
    $E' = E' \cup \{(v, i)\}$;
    if $l(i) < l(v) + 1$ **then**
        $l(i) = l(v) + 1$;
    end if
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**
        if $j \leq pn$ **then**
            choose $v_\ell \in B_j^\ell \cap Q \sim \{h_j^\ell\}$ of **MAXIMUM** $m(v_l)$ (or $l(v_l)$);
        else
            choose $v_\ell \in B_j^\ell \cap Q \sim \{h_j^\ell\}$ of **MINIMUM** $m(v_l)$ (or $l(v_l)$);
        end if
        PAR-STREC(G, v_ℓ);
    end for
end if

Note that PAR-STN(0) is equivalent to MIN-STN, thus $\Delta(0) = \lambda(t)$ while PAR-STN(1) is equivalent to MAX-STN and $\Delta(1) = \ell(t)$. PAR-STN 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 probability 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 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, there is a high probability that $\Delta(p) \leq p(n-1)$. In this case, $\Delta(p)$ is again very close to $p(n-1)$. As far as the parameterized *st*-orientation is concerned, we can extend our idea and insert more parameters $p_1, p_2, \ldots, p_k$. In this case the algorithm would compute a longest path equal to $\Delta(p_1, p_2, \ldots, p_k)$. 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. Algorithm 6 is the full pseudocode of the parameterized algorithm. Note that the algorithm can either use the *current timestamp* or the *longest path* timestamp method.

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

Chapter 4

Applications of Parameterized st-Orientations

4.1 General

The purpose of this work has always been the computation of st-oriented graphs with certain parameters, such as the *length of the longest path* from s to t. In this chapter we show that this research direction is indeed important for many applications.

4.2 Graph Drawing

4.2.1 General

During the past decades, a lot of algorithms have been proposed for drawing st-oriented graphs [3]. The structure of these graphs (actually the length of the longest path) is vital in the final drawing. Namely, the length of the longest path from s to t determines some of the most important features of the drawing, such as the width and the length of it, which are also dependent on the algorithm that is every time used in order to visualize the specific graphs.

It is also interesting to use the algorithms that apply to directed acyclic graphs as means to visualize general undirected graphs. In this case, we transform a general undirected graph G to a directed acyclic graph with a single source s and a single sink t, i.e., an st-oriented graph. This is feasible if and only if $G \cup \{s, t\}$ is biconnected [15]. The transformation can be achieved by computing an st-numbering (which implies an st-orientation or a bipolar orientation) of the respective graph G.

There are many linear time algorithms to compute bipolar orientations but clearly do not define some properties on the produced orientation [8, 7, 24]. Actually, they produce an orientation at random without a guarantee for the length of the longest path of the final st-oriented graph. Therefore, the presented algorithm could be used by many graph drawing algorithms and produce drawings of certain aesthetics. Following we refer to some graph drawing algorithms and techniques that are used for the visualization of st-oriented graphs and where the length of the longest path from s to t determines major characteristics of the final solution:

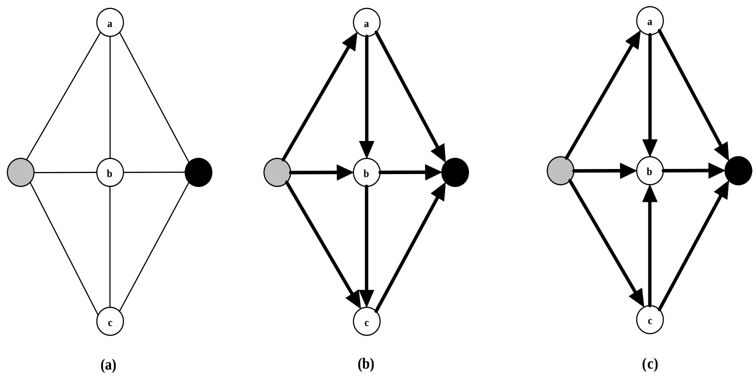


Figure 4.1: An undirected graph (a) and two (b), (c) possible st-orientations of it.

- **Hierarchical Drawings.** One of the most common algorithms in hierarchical drawing is the longest path layering [4]. This algorithm applies to directed acyclic graphs. The height of such a drawing is always equal to the length of the longest path of the directed acyclic graph, $l(t)$. If we want to visualize an undirected graph G using this algorithm, we must firstly st-orient G. The height of the produced drawing will be equal to the length of the longest path $l(t)$ of the produced st-orientation. By computing an st-orientation of the graph with low length of longest path ("*short st-orientations*"), we can produce drawings of low height. On the contrary if we want to produce maximal height drawings, we must compute an st-orientation of greater length of longest path ("*long st-orientations*"). This gives us the opportunity to produce drawings of desired sizes, according to the application.

- **Visibility Representations.** In order to compute visibility representations of planar graphs, we must compute an optimal topological numbering of an st-orientation of the input graph [22]. This can be done if we assign unit-weights to the edges of the graph and compute the longest path to each one of its vertices from source s. The longest path that each vertex is assigned is its optimal weighted topological number. The y-coordinate of each vertex in the visibility representation is equal to its topological number. Hence the length of the longest path of the used st-orientation is decisive in visibility representations of undirected graphs. Moreover, in the visibility representations, the length of the longest path $l^*(t)$ of the dual graph is also important.

How a different primal st-orientation impacts on the dual orientation is very crucial for visibility representations.

- **Orthogonal Drawings.** The first step of algorithms that compute orthogonal drawings [18] is to compute an st-numbering of the input undirected graph G. These algorithms compute some variables (such as the row pairs or the column pairs in [18]) that are functions of the st-orientation and which determine the width and the height of the drawing. Applying different st-orientations for the orthogonal drawing of a graph G, can result in different drawing area bounds.

Figure 4.1 depicts an undirected graph G (Figure 4.1a) and two different st-orientations of it. The length of the longest path (from s to t) of the first st-orientation (Figure 4.1b) is equal to 4, while the second st-orientation (Figure 4.1b) has length of longest path from s to t equal to 3. There is no other st-orientation of longest path length different than 3 or 4 for graph G. Figure 4.2 shows two different longest path and visibility representation layouts for the two different st-orientations (4.1b), (4.1c) of the same graph (4.1a). The drawings have different characteristics, for example drawings in Figure 4.2a are more "*longer*" and "*thinner*" whereas drawings in Figure 4.2b are "*shorter*" and "*wider*". Additionally, in the visibility representations layout, the required area is different. The length of the longest path of the dual graph also changes.

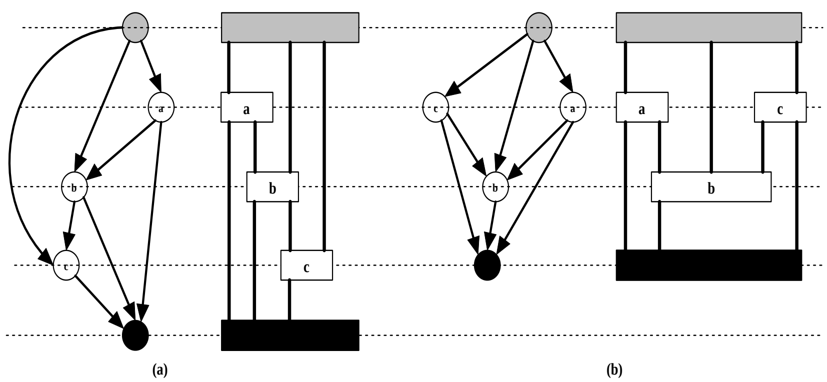


Figure 4.2: Longest path layering and visibility representation layouts for the st-orientation of Figure 4.1b (a) and for this of Figure 4.1c (b).

4.2.2 Primal and Dual st-Orientations

Now we present some results concerning the impact of parameterized st-orientations on st-planar graphs. As it is referred in [4], st-planar graphs G are planar graphs having two distinct nodes s, t on the outer face of their embedding. If we st-orient such a graph, we can define a single orientation for the dual graph G^* which is also an s^*t^*-orientation.

This method is used in the visibility representations algorithms [22], when we have to compute the dual s^*t^*-oriented graph. The length of the longest path of this graph determines the width of the geometric representation. Thus, the questions that arise are natural. What is the impact of the parameter p on the length of the longest path of the dual s^*t^*-oriented graph G^* of an st-planar graph G, which (the graph G) has been st-oriented with PAR-STN(p)? Intuitively, we would expect that $l^*(t)$ (the length of the longest path of the dual graph G^*) will grow inversely proportional to $l(t)$ (the longest path length of the primal graph G) (see Figure 4.3). As we will see, this is not always the case. In

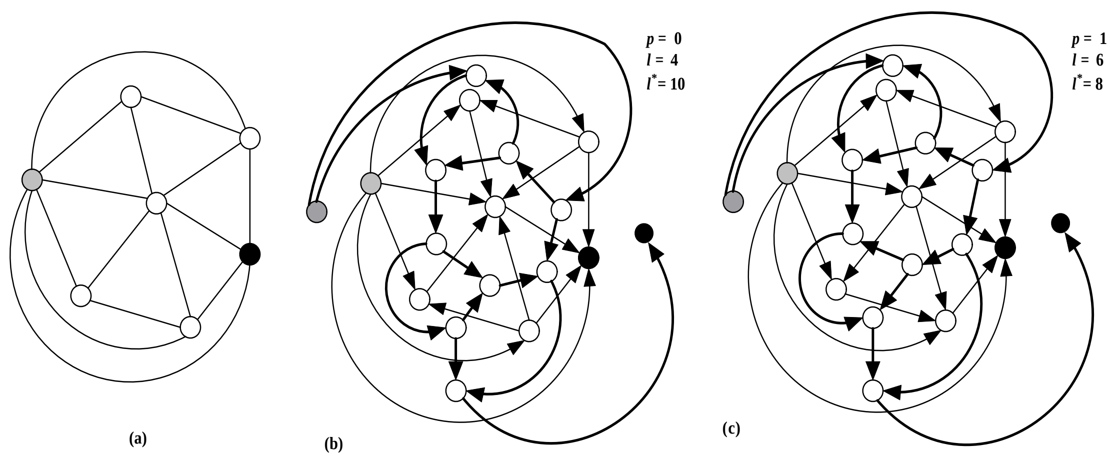


Figure 4.3: Constructing the dual graph for different values of the parameter p ($p = 0, 1$).

Figure 4.3, we can see the impact of the parameter p on the longest path length of the dual graph G^*. In 4.3a, an st-planar (undirected) graph G is shown. Note that this graph is triangulated and thus it has maximum density. In 4.3b, we construct an st-orientation of G, applying PAR-STN(0), getting $l(t) = 4$ and $l^*(t) = 10$. Note that $l(t) + l^*(t) = 14 = 2n$. Finally, in 4.3c we use PAR-STN(1) to get a primal st-orientation with $l(t) = 6$ and a dual st-orientation with $l^*(t) = 8$. In this case $l(t) + l^*(t) = 14 = 2n$.

4.2.3 A Special Class of Planar Graphs

In this section we investigate certain classes of st-planar graphs that can be st-oriented in such a way that certain lengths of primal and dual longest paths can be achieved. This is actually a good reason to justify the fact that different st-orientations are indeed important in many applications.

Definition 4.2.1. *We define an n-path planar graph ($n \geq 5$) $G = (V, E)$ to be the planar graph that consists of a path $P = v_2, v_3, \ldots, v_{n-1}$ of $n-2$ nodes and two other nodes v_1, v_n such that $(v_1, v_i) \in E$, $(v_i, v_n) \in E$ $\forall i = 2, \ldots n-1$ and $(v_1, v_n) \in E$.*

In Figure 4.4, one n-path planar graph is depicted. Its source is node 1 whereas its sink is node $n-1$. Note that an $(n+1)$-path planar graph G_{n+1} can be obtained from an n-path planar graph G_n if we add a new node and connect it with nodes v_1, v_2 and v_n (nodes v_1 and v_n are the rightmost and leftmost nodes of G_n's embedding in Figure 4.4). Let now

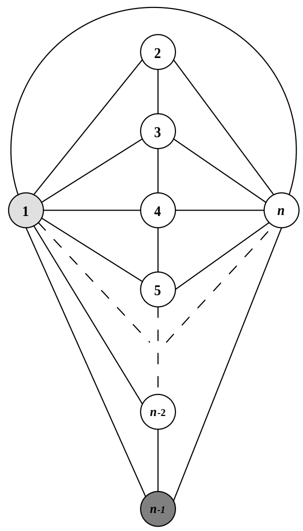


Figure 4.4: An n-path planar graph. We define node 1 to be the source of the graph and node $n-1$ to be the sink of the graph.

G_n be an n-path planar graph and $\lambda(G_n), \ell(G_n)$ denote the minimum and the maximum longest path length $1(n-1)$-orientations over the set of all the $1(n-1)$-orientations of G_n respectively. In Figure 4.5, two $1(n-1)$-orientations of a general n-path planar graph are depicted. In Figure 4.5a, the orientation of minimum longest path length is depicted while in Figure 4.5b the orientation of maximum longest path length is depicted. Let $G_1(n)$ be the directed graph of Figure 4.5a and $G_2(n)$ be the directed graph of Figure 4.5b. Note the difference between the two orientations of Figure 4.5. In 4.5a, all edges belonging on the medium path of $G_1(n)$ successively change their direction. On the other hand, in 4.5b, all edges belonging on the medium path of $G_2(n)$ have an orientation towards the sink of the graph, $n-1$.

In the following Lemmas, we say that a graph is a minimum (maximum) st-oriented graph if it is st-oriented and the length of the longest path from s to t is the minimum (maximum) over all the possible st-orientations of the respective undirected graph.

Lemma 4.2.2. *For all $n \geq 5$ $G_1(n)$ is one minimum $1(n-1)$-oriented graph of an n-path planar graph G_n. Moreover, for this orientation, it is $\lambda(G_n) = 4$.*

Proof. We will use induction. For $n = 5$ examine all the 1-4 orientations (2^3 orientations minus the cyclic and the multiple source or multiple sink ones) of G_5 and we have $\lambda(G_5) = 4$

which is achieved with the st-oriented graph $G_1(5)$. Suppose the Lemma holds for $n = k$. Hence the minimum $1(k-1)$-oriented graph is $G_1(k)$ and $\lambda(G_k) = 4$. For $n = k+1$ construct a directed graph H from $G_1(k)$ by inserting a new node v and adding the directed edge $(v, 2)$ if n is odd or $(2, v)$ if n is even and the directed edges $(1, v)$ and (v, n). Then H is the graph $G_1(k+1)$ and the longest path length does not increase due to the way v is connected to node 2, as the newly added edge can either replace or not an edge that participated in $G_1(k)$'s longest path. Additionally, the addition of node v does not create any cycle as it adds two cycle-free triangles to an already st-oriented graph. Hence $\lambda(G_{k+1}) = 4$ and G_{k+1} is $1(k)$-oriented, which implies the correctness of the Lemma. □

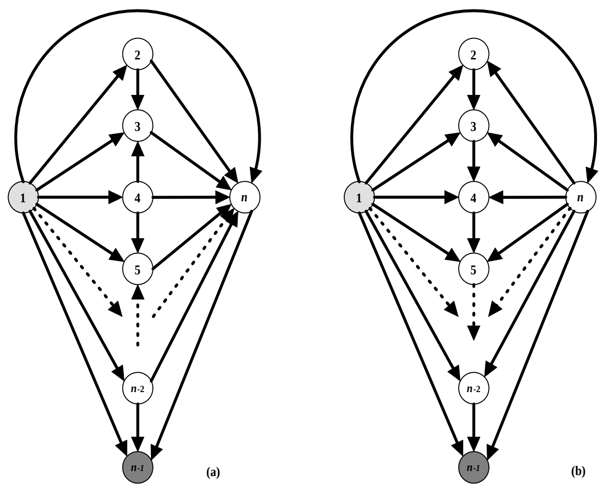


Figure 4.5: Two $1(n-1)$-orientations of an n-path planar graph, $G_1(n)$ (a) and $G_2(n)$ (b).

Lemma 4.2.3. *For all $n \geq 5$ $G_2(n)$ is one maximum $1(n-1)$-oriented graph of an n-path planar graph G_n. Moreover, for this orientation, it is $\ell(G_n) = n - 1$.*

Proof. The proof is again by induction as in Lemma 4.2.2. The only difference here is that the addition of a new node each time increments the primal longest path length. □

Suppose now we compute the dual graphs of the directed graphs $G_1(n)$ and $G_2(n)$, $G_1^*(n)$, $G_2^*(n)$ respectively (see Figure 4.6a and 4.6b respectively). Let $\lambda^*(G_n)$, $\ell^*(G_n)$ denote their respective longest path lengths.

Theorem 4.2.4. *For all $n \geq 5$ it holds $\lambda^*(G_n) = \ell^*(G_n) = 2n - 4$.*

Proof. We will use induction. For $n = 5$, compute the dual orientations $G_1^*(5)$, $G_2^*(5)$ of the respective directed graphs $G_1(5)$, $G_2(5)$. Then it holds $\lambda^*(G_5) = \ell^*(G_5) = 6 = 2 \times 5 - 4$. Suppose the Lemma holds for $n = k$, i.e., $\lambda^*(G_k) = \ell^*(G_k) = 2k - 4$. For $n = k + 1$, add a node v (node $k+1$ in Figure 4.7) as in previous Lemmas to construct the primal directed graphs $G_1(k+1)$ and $G_2(k+1)$ and then construct the dual orientations of them as in Figure 4.7. In both cases, 3 new dual edges are introduced (the thick black edges). Only two of them participate in the longest path lengths $\lambda^*(G_{k+1})$, $\ell^*(G_{k+1})$ and therefore the longest path lengths $\lambda^*(G_{k+1})$, $\ell^*(G_{k+1})$ are increased by two. Hence:

$$\lambda^*(G_{k+1}) = \lambda^*(G_k) + 2 = 2k - 4 + 2 = 2(k+1) - 4$$

and

$$\ell^*(G_{k+1}) = \ell^*(G_k) + 2 = 2k - 4 + 2 = 2(k+1) - 4$$

which entail that $\lambda^*(G_n) = \ell^*(G_n) = 2n - 4$. □

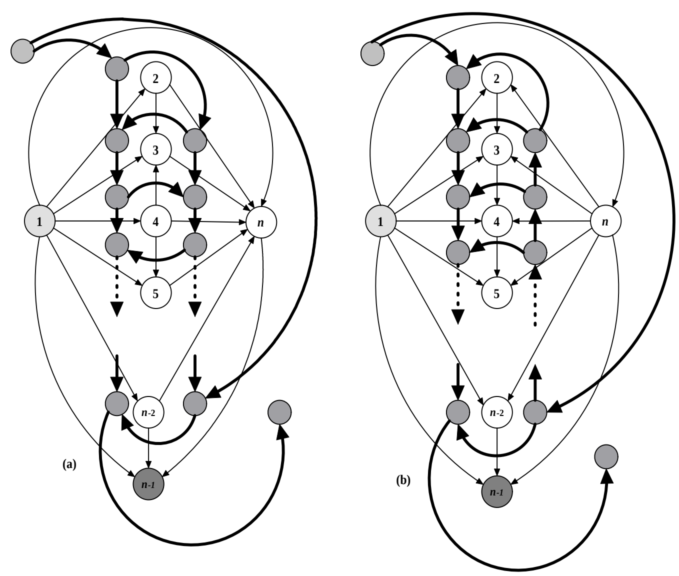


Figure 4.6: The dual orientations of $G_1(n)$ (a) and $G_2(n)$ (b).

According to Theorem 4.2.4, the impact of different st-orientations of an n-path planar graph on the area of their visibility representation is evident. By using the minimum st-orientation, we will need an area equal to

$$\lambda(G_n)\lambda^*(G_n) = 4(2n - 4) = 8n - 16 = O(n)$$

If we use the maximum st-orientation, we will need an area equal to

$$\ell(G_n)\ell^*(G_n) = (n-1)(2n-4) = 2n^2 - 6n + 4 = O(n^2)$$

In this way, we can reduce the area by a factor of n. Note that while $\ell(G_n) + \ell^*(G_n) = 3n - 5 > 2n$, it is $\lambda(G_n) + \lambda^*(G_n) = 2n \leq 2n$. We therefore introduce the following conjecture:

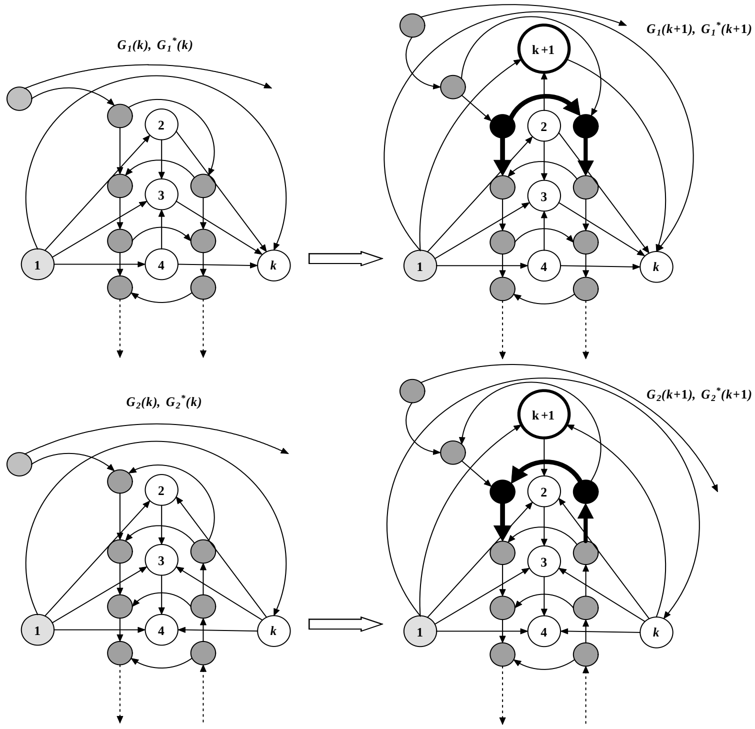


Figure 4.7: The inductive step of the Theorem 4.2.4. The length of the dual longest path always increases by two.

Conjecture 4.2.5. *For every n-node planar biconnected graph G, two nodes s, t of its vertex set, there exists at least one st-orientation of G such that $l(t) + l^*(t) \leq 2n$.*

In order to face this conjecture, one should try to devise an algorithm that deterministically st-orients a planar graph in a way that the produced length of the dual longest path grows at most as much as the primal one does.

As we will see in the experimental results section, the inequality $l + l^* \leq 2n$ holds for the majority of the planar graphs tested. Actually, for all the planar graphs tested, STN always computes an st-orientation that satisfies this inequality.

4.3 Longest Path Problem

As we saw in the previous chapter, computing an st-orientation of maximum longest path length is an NP-hard problem (reduction from the directed Hamilton Problem). However, 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 (for st-Hamiltonian graphs it computes an st-orientation of longest path length roughly equal to $n-1$.)

4.4 Graph Coloring Problem

In the Computational Complexity section of the previous chapter, we proved that computing an st-orientation of minimum longest path length is NP-hard. To do so, we reduced the Graph Coloring Problem to it.

Theorem 3.5.4 shows a strong connection of graph coloring and minimum longest path length st-orientations. By producing a good solution for the minimum st-orientation prob-

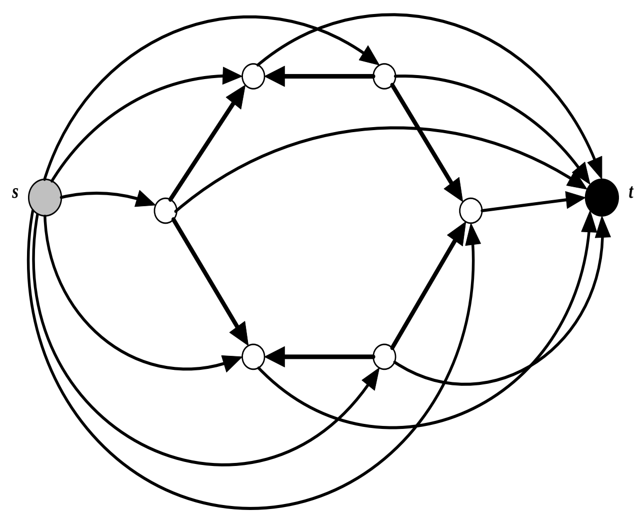


Figure 4.8: Combining graph coloring and st-orientations.

lem we maybe have a good solution for the graph coloring problem. Suppose we are given a graph $G=(V,E)$ and we want to compute a coloring of G. We produce the graph $G'=(V',E')$ by adding two extra nodes s, t and edges from s to all the nodes of G and from t to all the nodes of G. We apply the MIN-STN algorithm to G' with source s and sink t, resulting to an st-oriented graph F with longest path l. Then, by Theorem 3.5.4, we can color G using $l-1$ colors.

We illustrate this thought with an example. Suppose we want to compute a coloring of a ring $G=(V,E)$ consisting of 6 nodes. Clearly $\chi(G)=2$. If we add the nodes s, t, the undirected edges (s,i), (t,i) $\forall i \in V$ and apply MIN-STN to it, we produce the st-oriented graph of Figure 4.8.

Note that all nodes lying on the ring have a longest path length from s either 1 or 2. The longest path length from s to t is 3, and thus we need $3-1=2$ colors to color G. Actually, this is the chromatic number of G. Hence, we have computed the chromatic number of G by applying MIN-STN to G'. The question that arises is whether MIN-STN can compute good colorings for other graphs as well. This is something that opens new research directions (maybe the development of new heuristics adjusted to the graph coloring problem in order to break the ties that appear in the choice of the next source during MIN-STN) and has to be tested experimentally.

Theorem 3.5.4 is also very important in Graph Drawing. If one could prove that a minimum longest path length st-orientation implies an optimal drawing area of a layout of a planar graph, then, by Theorem 3.5.4 it would be easy to conclude that computing such an optimal layout would be NP-hard.

4.5 st-Orientations of Special Classes of Graphs

Following, we investigate st-orientations of special classes of graphs. The special structure of some graphs can lead us to useful conclusions concerning their st-orientations. Additionally, the structure of some graphs does not allow the existence of many st-orientations and hence the impact of the parameter is not clear. In this section we investigate these *hard* classes of graphs and prove some properties.

4.5.1 st-TSA graphs

In this section, we describe a special case of graphs that can be constructed by using the algorithm that 2-approximates the metric TSP (the TSP where the triangle inequality is satisfied) problem. Actually this method constructs st-Hamiltonian graphs of low density (always less than $2-\frac{3}{n}$). These graphs have very important properties that clearly influence the final orientation and the length of the longest path of the final st-oriented graph. As

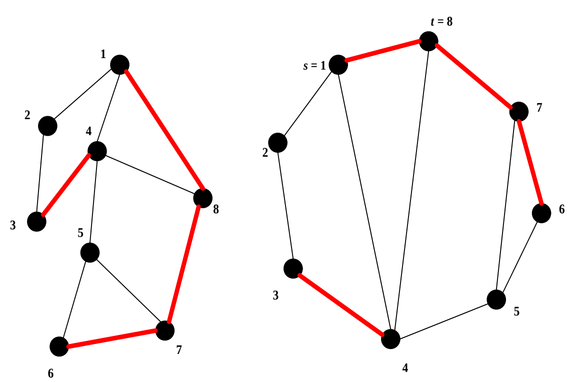


Figure 4.9: Constructing an st-TSA graph.

is widely known, Christophides [6] devised a $\frac{3}{2}$-approximation algorithm for the well known NP-complete metric TSP problem. This method is an improvement of the algorithm that uses preorder numbering and shortcuts in order to achieve a 2-approximation of the metric TSP problem. We will use the last algorithm in order to construct an st-Hamiltonian graph of density $2 - \frac{3}{n}$ or less. We call these graphs st-TSA graphs (st-Travelling Salesman Approximation Graphs). Following, we present the algorithm we have used in detail (Figure 4.9).

Suppose we want to construct an n-node st-TSA graph. We initialize a K_n graph with random weights on its edges. Then we compute a minimum spanning tree $T' = (V, E)$ of it by using either the Kruskal or the Prim algorithm.

Afterwards we pick a node $r \in V$ to be the root of the tree T' and we execute a preorder traversal of the tree T' from the root r. Thus every node v_i gets a preorder number $d(v_i)$ such that

$$d(v_i) < d(v_j)$$

if v_i occurs before v_j in the preorder numbering. Suppose we store the nodes of the tree in a vector y sorted in increasing preorder number. Note that $y_1 = r$. We now expand the minimum spanning tree T by setting

$$E_s = E \cup \left(\bigcup_{i=1}^{n-1} (y_i, y_{i+1}) \right) \cup (y_n, y_1)$$

It is clear now that $|E_s| \leq 2n - 3$ (tight in the case of a minimum spanning tree where all nodes except for the root r have the same father r) and hence the density bound follows. Additionally, there is always a Hamilton path that connects all the nodes of the graph from r to y_n. If we now set $s = r = y_1$ and $t = y_n$, we get an st-Hamiltonian graph $G = (V, E_s)$ of density at most $2 - \frac{3}{n}$. We call this graph an st-TSA graph.

Following we describe some properties concerning the st-TSA graphs and play a major role in the st-orientation. Actually, we refer to the class of outerplanar graphs. Outerplanar graphs are a subclass of planar graphs with the additional property that all nodes can be placed on a circle circumference in a way that an embedding with zero crossings is produced.

Theorem 4.5.1. *st-TSA graphs are outerplanar.*

Proof. Let $G = (V, E)$ be an n-node st-TSA graph. Let y be the preorder vector with $y_1 = s$ and $y_n = t$. We want to prove that if we place all nodes with the order they appear in the preorder vector on a circle circumference, an embedding with zero crossings is produced. Suppose there is at least one crossing. Then there will exist at least one quadruple of integers (k, i, j, l) such that $k < i < j < l \leq n$ such that

$$(y_k, y_j) \in E \wedge (y_i, y_l) \in E$$

The crossing x defined by this quadruple will be the intersection point of the line segments $[y_k, y_j]$ and $[y_i, y_l]$. (Figure 4.10a). Note that both edges (y_k, y_j) and (y_i, y_l) should be tree edges because all added edges must lie on the circumference of the circle and could not create a crossing.

The edge (y_k, y_j) is a tree edge and from hypothesis it is $k < j$. As $k < i < j$, y_i is discovered before y_j. Hence it is (y_i) the root of a dfs subtree on the left of y_j (all the nodes on the left of y_j must have less preorder number than the preorder number of y_j) (Figure 4.10b). The edge (y_i, y_l) is also a tree edge. Therefore y_l must lie on the left of y_j, as its tree father y_i lies on the left of y_j. This means that y_l lies before y_j in the preorder numbering, i.e., $l < j$, which does not hold as from hypothesis we have that $k < i < j < l \leq n$. Hence there is no crossing x and therefore G is outerplanar. □

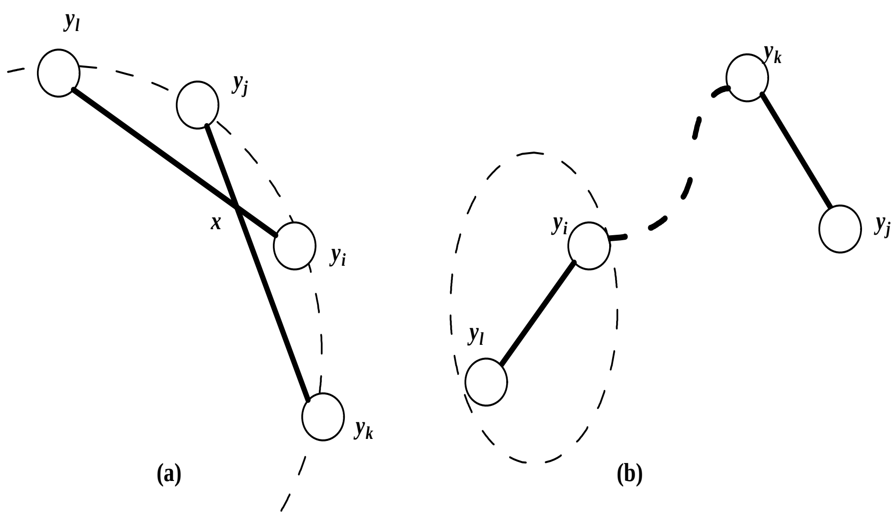


Figure 4.10: Proof of theorem 4.5.1.

4.5.2 Series-Parallel Graphs and Outerplanar Graphs

Following we refer to a special case of graphs called series-parallel graphs and their orientations. We are going to show relation between outerplanar graphs and series-parallel graphs. A series-parallel graph $G_{s,t}$ is recursively defined as follows:

- **Base Case:** A graph G consisting of two nodes s, t connected by an edge is a series parallel graph. Nodes s, t are called source and sink of G respectively. Let G_1 and G_2 be two series parallel graphs with sources s_1, s_2 and sinks t_1, t_2 respectively.

- **Serial Combination:** The graph G_s that emerges from identifying t_1 with s_2. The source of G_s is s_1 and the sink of G_s is t_2.

- **Parallel Combination:** The graph G_p that emerges from identifying s_1 with s_2 and t_1 with t_2. The source of G_p is s_1/s_2 and the sink is t_1/t_2.

Note that a series parallel graph admits a single st-orientation. This is so because any series parallel graph can be decomposed into base-case series parallel graphs (two nodes s, t and one connecting edge (s,t)) which all admit a single st-orientation from s to t. Hence:

Theorem 4.5.2. *Let G be a series-parallel graph with source s and sink t. Then G admits exactly one st-orientation.*

Following, we must make a very important remark. All biconnected outerplanar graphs (outerplanar graphs with a Hamilton cycle, such as the st-TSA graphs) are series-parallel graphs if and only if the source s and the sink t are carefully chosen to be two nodes that satisfy certain properties.

Let $G = (V,E)$ be a biconnected outerplanar graph. Then all nodes of G lie on the circumference of a cycle according to a certain ordering. We now define two relations on the nodes of G. For every two nodes $u, v \in V$ we say that $u \rightarrowtail v$ if and only if u, v are cycle-adjacent, i.e., they lie in neighboring positions on the circle of G and $(u,v) \in E$. In similar, we say that $u \rightarrow v$ if and only if u, v are out-adjacent, i.e., they lie in non-neighboring positions on the circle of G and $(u,v) \in E$. We can now divide the set of edges E of each biconnected outerplanar graph in two subsets E_c and E_o such that

$$E_c = \{(u,v) \in E : u \rightarrowtail v\}$$

and

$$E_o = \{(u,v) \in E : u \rightarrow v\}$$

where $E = E_c \cup E_o$. Suppose now we choose two random nodes s, t on the circle of G. This random choice defines two paths $\Pi_{(s,t)}$, $\Pi'_{(s,t)}$ on the circumference of the circle (see Figure 4.11).

For each biconnected outerplanar graph with one distinct source s and one distinct sink t, we define an edge set $C_{(s,t)} \subseteq E_o$ such that

$$C_{(s,t)} = \{(u,v) \in E_o : u \in \Pi_{(s,t)} \wedge v \in \Pi'_{(s,t)}\}$$

Lemma 4.5.3. *Let G be a biconnected outerplanar graph and s, t be two nodes of it. G is a series-parallel graph with source s and sink t if and only if $C_{(s,t)} = \{\emptyset\}$*

Proof. For the direct, suppose G is a series-parallel graph with source s and sink t and $C_{(s,t)} \neq \{\emptyset\}$. Then an edge $(v_i, v_j) \in C_{(s,t)}$ splits G into two subgraphs G_l (which contains s) and G_R (which contains t). If we now execute a parallel combination of G_l, G_R (which

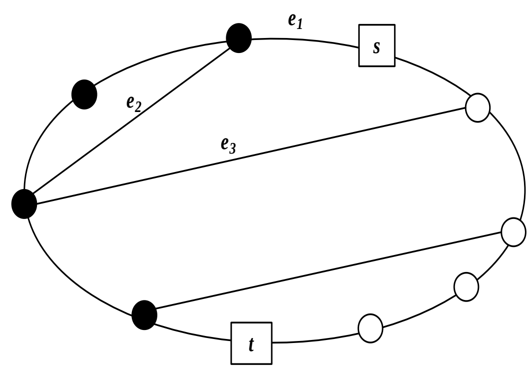


Figure 4.11: Choosing two nodes s, t defines two paths Π (white vertices), Π' (black vertices). $C_{(s,t)}$ contains edges with different color endpoints.

are both series-parallel of source v_i and sink v_j), we get a series-parallel graph of source v_i and sink v_j. This series-parallel graph is G. But G has source s and sink t, hence our initial assumption does not hold and $C_{(s,t)} = \{\emptyset\}$.

For the inverse, we must prove that if $C_{(s,t)} = \{\emptyset\}$ then G is a series-parallel graph with source s and sink t. As $C_{(s,t)} = \{\emptyset\}$ all the edges $e = (v_i, v_j)$ of the graph have both their endpoints either on $\Pi_{(s,t)}$ or on $\Pi'_{(s,t)}$. Without loss of generality, let $v_i, v_j \in \Pi_{(s,t)}$. Then there exist $k_1, k_2, \ldots, k_l$ such that $v_i \rightarrowtail k_1, k_1 \rightarrowtail k_2, \ldots, k_l \rightarrowtail v_j$. This path is actually a series-parallel graph with source v_i and sink v_j, which, if combined (in parallel) with the edge $e = (v_i, v_j)$ gives a series-parallel of source v_i and sink v_j, which we call a *gadget*. If this operation is performed for all such edges, we can serially combine all gadgets and remaining edges on the circle and get two series parallel graphs G_Π and $G_{\Pi'}$ (one for each path $\Pi_{(s,t)}$ and $\Pi'_{(s,t)}$). Both G_Π, $G_{\Pi'}$ have source s and sink t. A parallel combination of G_Π, $G_{\Pi'}$ gives the desired result. Hence G is a series-parallel graph with source s and sink t. $\square$

By Theorem 4.5.2 and Lemma 4.5.3 we get:

Corollary 4.5.4. *Let G be a biconnected outerplanar graph and s, t be two nodes of it. If $C_{(s,t)} = \{\emptyset\}$, then G admits a single st-orientaton.*

Corollary 4.5.4 implies that *st*-TSA graphs also admit a single *st*-orientation.

Chapter 5

Experimental Results

5.1 General

In this chapter, we present our experimental results produced by executing the presented *parameterized* algorithms on various classes of graphs. It will be made clear to the reader that the algorithms perform exceptionally good on the used graphs and finally achieve their mission (and initial purpose of their development), which is the *influence* of the **length** of the longest path of the final st-oriented graph. All the presented experiments were run on a Pentium IV processor of 512 MB RAM running at 2.8 GH under Windows 2000 Professional (entpc2 machine in the C031 room at the basement of the Computer Science Department of the University of Crete). The algorithms were implemented in Java v.1.4, using the Java Data Structures Library (http://www.jdsl.org) [11].

5.2 st-Hamiltonian Graphs

The first tests were conducted on st-Hamiltonian Graphs. 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.

5.2.1 Construction of Graphs

In order to construct the graphs in 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 $(P(1), P(2)), (P(2), P(3)), \ldots, (P(n-1), P(n)), (P(n), P(1))$ 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

Table 5.1: Results for parameterized *st*-orientations of density 2.5 *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	16.20	0.164	43.10	0.435	59.50	0.601	75.80	0.766	93.00	0.939
200	20.50	0.103	74.70	0.375	111.60	0.561	146.30	0.735	183.10	0.920
300	25.80	0.086	107.00	0.358	163.00	0.545	212.20	0.710	275.30	0.921
400	27.70	0.069	139.20	0.349	216.60	0.543	287.40	0.720	368.30	0.923
500	28.50	0.057	170.30	0.341	263.30	0.528	358.60	0.719	458.80	0.919
600	30.60	0.051	199.30	0.333	314.10	0.524	422.20	0.705	553.50	0.924
700	33.80	0.048	232.40	0.332	364.70	0.522	492.20	0.704	644.90	0.923
800	36.90	0.046	266.80	0.334	416.00	0.521	555.80	0.696	736.70	0.922
900	37.50	0.042	294.70	0.328	462.00	0.514	627.30	0.698	834.60	0.928
1000	38.10	0.038	324.60	0.325	515.40	0.516	694.50	0.695	924.20	0.925
1100	38.10	0.035	356.80	0.325	567.60	0.516	766.40	0.697	1010.30	0.919
1200	37.80	0.032	388.00	0.324	616.10	0.514	834.20	0.696	1108.80	0.925
1300	46.70	0.036	416.80	0.321	665.70	0.512	904.90	0.697	1197.00	0.921
1400	40.20	0.029	450.90	0.322	714.00	0.510	971.40	0.694	1291.30	0.923
1500	48.90	0.033	479.20	0.320	769.80	0.514	1038.90	0.693	1387.90	0.926
1600	50.70	0.032	508.80	0.318	815.60	0.510	1111.20	0.695	1478.30	0.925
1700	43.00	0.025	541.20	0.319	864.80	0.509	1176.00	0.692	1583.00	0.925
1800	41.80	0.023	571.30	0.318	912.70	0.507	1245.10	0.692	1667.30	0.927
1900	47.60	0.025	603.30	0.318	965.60	0.508	1311.40	0.691	1750.30	0.922
2000	50.8	0.025	629.40	0.315	1016.40	0.508	1378.80	0.690	1847.00	0.924

Table 5.2: 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

Table 5.3: 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 5.4: 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

Table 5.5: Results for parameterized st-orientations of density 6.5 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	14.30	0.144	41.40	0.418	58.90	0.595	76.70	0.775	95.20	0.962
200	16.90	0.085	74.10	0.372	113.00	0.568	147.90	0.743	191.70	0.963
300	20.10	0.067	103.70	0.347	162.20	0.542	220.10	0.736	287.20	0.961
400	20.30	0.051	136.20	0.341	215.30	0.540	291.70	0.731	384.50	0.964
500	23.60	0.047	168.20	0.337	265.00	0.531	362.40	0.726	482.60	0.967
600	24.40	0.041	198.90	0.332	318.40	0.532	434.70	0.726	576.80	0.963
700	27.60	0.039	229.90	0.329	369.20	0.528	506.30	0.724	677.80	0.970
800	26.70	0.033	263.60	0.330	421.20	0.527	576.10	0.721	769.00	0.962
900	25.70	0.029	293.20	0.326	470.50	0.523	647.70	0.720	869.60	0.967
1000	26.90	0.027	324.40	0.325	520.10	0.521	715.10	0.716	966.00	0.967
1100	27.70	0.025	355.40	0.323	575.60	0.524	785.30	0.715	1063.60	0.968
1200	29.20	0.024	385.80	0.322	625.00	0.521	860.60	0.718	1157.20	0.965
1300	33.10	0.025	414.90	0.319	674.00	0.519	927.80	0.714	1253.60	0.965
1400	34.20	0.024	445.40	0.318	720.80	0.515	998.20	0.714	1348.50	0.964
1500	30.00	0.020	478.80	0.319	772.60	0.515	1069.10	0.713	1447.70	0.966
1600	32.70	0.020	509.10	0.318	823.90	0.515	1139.10	0.712	1541.40	0.964
1700	31.50	0.019	536.60	0.316	873.30	0.514	1210.00	0.712	1643.00	0.967
1800	36.50	0.020	566.70	0.315	924.20	0.514	1277.30	0.710	1738.50	0.966
1900	35.60	0.019	597.40	0.315	977.60	0.515	1349.90	0.711	1831.70	0.965
2000	37.2	0.019	627.50	0.314	1026.80	0.514	1420.10	0.710	1928.00	0.964

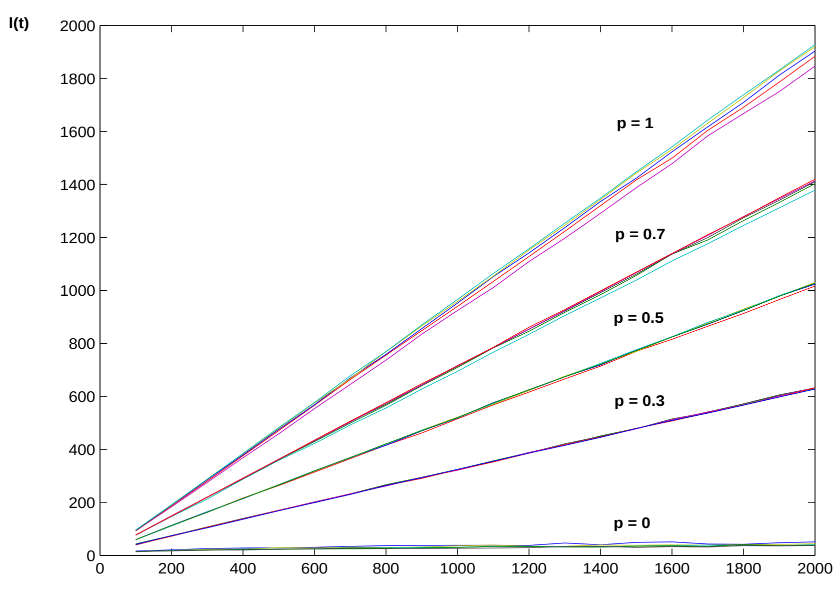


Figure 5.1: Length of longest path as a function of the parameter p for various graph sizes n and various density values d.

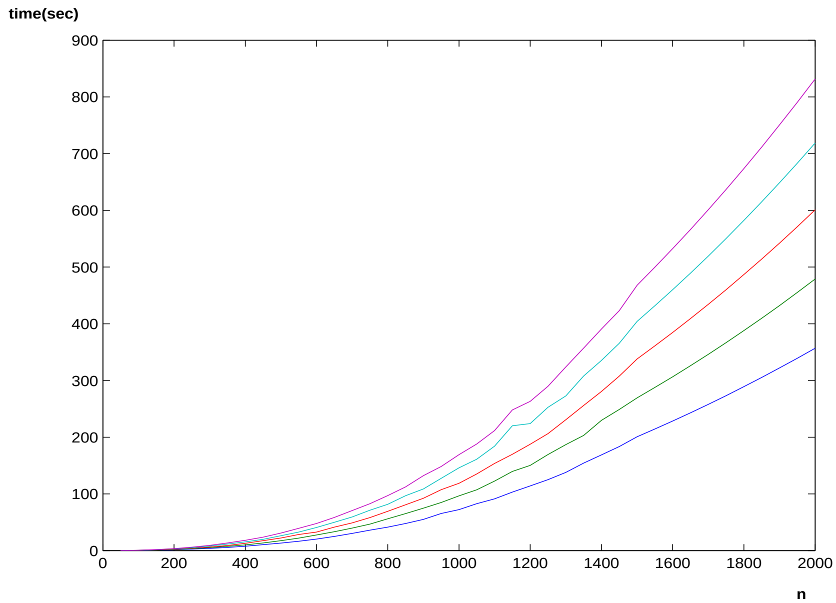


Figure 5.2: Execution time for various graph sizes n and various density values d.

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.

5.2.2 Computational Results

From Tables 5.1, 5.2, 5.3, 5.4, 5.5 and Figure 5.1 we can see that there is indeed a big influence of the parameter p on the length of the longest path of the final st-oriented graph. Actually, for a value $p = p_0$ the length of the longest path length of the produced st-oriented graph is roughly $p_0(n-1)$. Note that 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.

Following we justify why PAR-STN(p) computes st-oriented graphs of longest path length roughly equal to $p(n-1)$. To do this, we use the presented experimental results and support that $\Delta(1) = \Omega(n)$ and $\Delta(0) = \Omega(1)$ as for $p = 1$ the induced longest path length is at least $0.9n$ and for $p = 0$ the induced longest path length is almost constant. Suppose now we apply PAR-STN(p) to an st-Hamiltonian graph G. This means that we apply PAR-STN(1) (i.e., MAX-STN) for the first $\lceil pn \rceil$ iterations and PAR-STN(0) (i.e., MIN-STN) for the remaining $\lfloor (1-p)n \rfloor$ iterations.

During PAR-STN(1), suppose the nodes $v_1, v_2, \ldots, v_{\lceil pn \rceil}$ are removed. Let $G(1)$ be the subgraph of G that includes $v_1, v_2, \ldots, v_{\lceil pn \rceil}$. PAR-STN(1) is totally applied on $G(1)$ and thus gives an orientation of longest path length equal to $\Omega(pn)$, given that $\Delta(1) = \Omega(n)$.

During PAR-STN(0) the remaining nodes $v_{\lceil pn \rceil+1}, v_{\lceil pn \rceil+2}, \ldots, v_n$ are removed. In the same way, let $G(0)$ be the subgraph of G that includes these vertices. PAR-STN(0) is totally applied on $G(0)$ and thus gives an orientation of longest path length equal to $\Omega(1)$, given that $\Delta(0) = \Omega(1)$.

As the final length of longest path will roughly be the sum of the lengths of longest path of each one of the graphs $G(0)$, $G(1)$, we get that the approximate value for the total longest path length will be

$$\Omega(pn) + \Omega(1) = \Omega(pn)$$

Note that the above result is totally based on experimental results. In fact, PAR-STN(1) is an experimental constant approximation algorithm for the longest path problem. Finding a theoretical constant approximation algorithm for this problem has been proved to be NP-hard [25].

Finally, in Figure 5.2, the execution time of the algorithm is shown. The algorithm clearly runs in quadratic time ($O(nm) = O(dn^2)$) which depends on the density of the graph.

5.3 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 verify that the parameter works in a very efficient way for this class of graphs as well.

5.3.1 Construction of the Graphs

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 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.

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.[1]. This software produces graphs in ascii format which are easily transformed to an input for our algorithm.

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

Table 5.6: 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

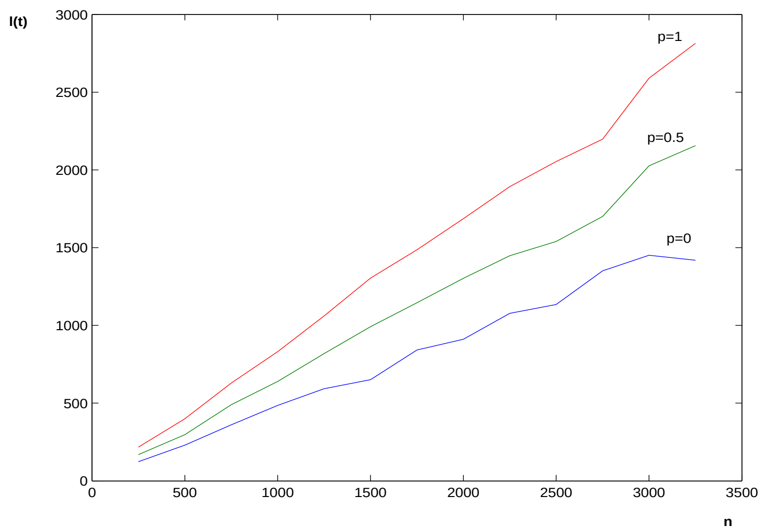
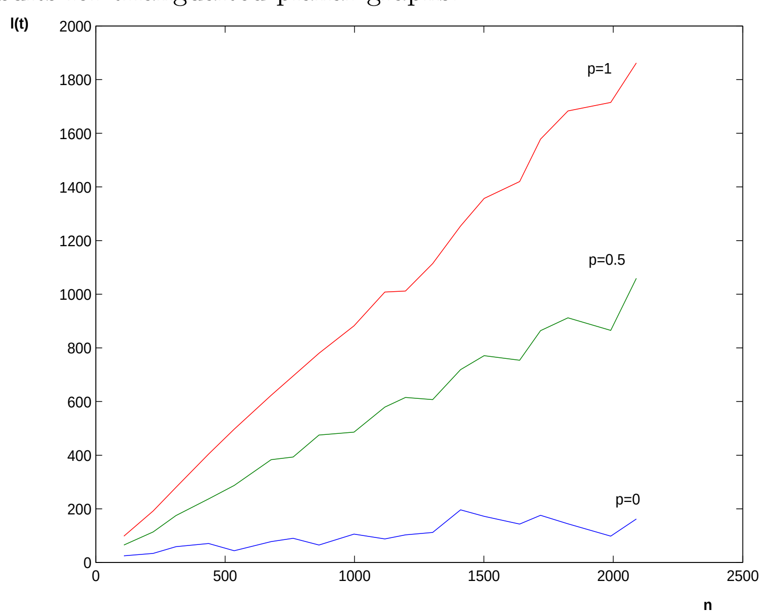


Table 5.7: Results for triangualted planar graphs.

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



5.3.2 Computational Results

In Tables 5.6, 5.7 the results for low density planar graphs and triangulated planar graphs are presented. Note that the effect of the parameter is again evident. For low density planar graphs (Table 5.6), 10 graphs of the same size were again tested and the mean of the longest path length is finally presented.

5.3.3 Visibility Representations

As we saw in the previous chapter, when we want to compute a visibility layout of a planar graph G, we must compute the dual graph of it [22]. The area of the computed layout is wholly dependent on the length of the primal longest path $l(t)$ and the length of the dual longest path $l^*(t)$ (actually it is the product $l(t) \times l^*(t)$). In this section we present the impact of the parameter p on the length of the dual longest path which finally translates into savings in the area of visibility layouts and justifies the importance of parameterized st-orientations in Graph Drawing applications.

Table 5.8: Primal and dual longest path length for low density st-planar graphs.

		p=0			**p=0.5**			**p=1**			$l \times l^*$		
n	$2n$	l	l^*	$l+l^*$	l	l^*	$l+l^*$	l	l^*	$l+l^*$	**p=0**	**p=0.5**	**p=1**
100	200	54	31	85	62	30	92	75	19	94	1674	1860	1425
200	400	111	36	147	138	26	164	173	14	187	3996	3588	2422
300	600	149	39	188	199	32	231	251	20	271	5811	6368	5020
400	800	190	112	302	257	81	338	346	19	365	21280	20817	6574
500	1000	165	129	294	339	73	412	454	16	470	21285	24747	7264
600	1200	302	118	420	378	120	498	462	32	494	35636	45360	14784
700	1400	412	208	620	502	130	632	626	17	643	85696	65260	10642
800	1600	447	156	603	565	156	721	717	19	736	69732	88140	13623
900	1800	396	178	574	501	108	609	664	32	696	70488	54108	21248
1000	2000	619	188	807	757	118	875	884	41	925	116372	89326	36244
1100	2200	438	287	725	649	221	870	841	31	872	125706	143429	26071
1200	2400	596	283	879	832	196	1028	1014	43	1057	168668	163072	43602
1300	2600	756	361	1117	970	182	1152	1150	34	1184	272916	176540	39100
1400	2800	599	497	1096	1010	315	1325	1260	29	1289	297703	318150	36540
1500	3000	835	345	1180	1047	281	1328	1281	46	1327	288075	294207	58926
1600	3200	617	599	1216	865	313	1178	1407	36	1443	369583	270745	50652
1700	3400	671	327	998	963	296	1259	1100	44	1144	219417	285048	48400
1800	3600	926	499	1425	1258	292	1550	1635	32	1667	462074	367336	52320
1900	3800	681	685	1366	1241	333	1574	1536	35	1571	466485	413253	53760
2000	4000	1147	337	1484	1503	239	1742	1803	44	1847	386539	359217	79332
2500	5000	1010	712	1722	1471	511	1982	2146	33	2179	719120	751681	70818
3000	6000	1652	683	2335	2114	555	2669	2608	44	2652	1128316	1173270	114752
3500	7000	1486	695	2181	2804	695	3499	2804	49	2853	1032770	1948780	137396
4000	8000	1500	1115	2615	2271	745	3016	3619	50	3669	1672500	1691895	180950
5000	10000	2101	1358	3459	2500	763	3263	3482	59	3541	2853158	1907500	205438

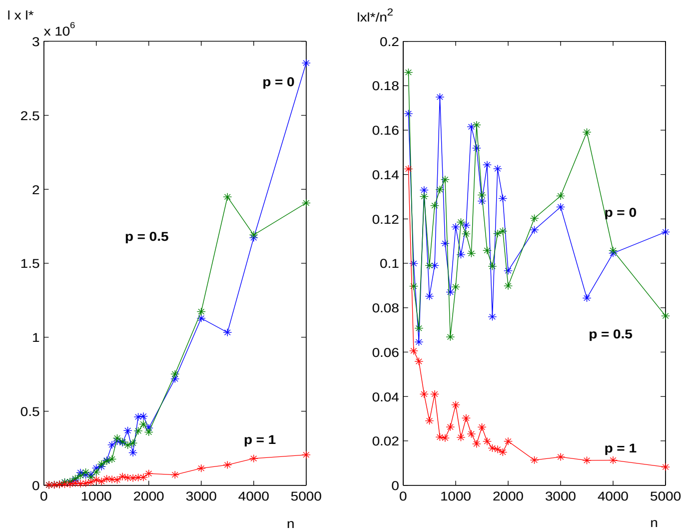


Figure 5.3: Absolute (left) and normalized (divided by n^2) (right) results for visibility representation area requirement for different values of the parameter p and low density planar graphs.

Table 5.9: Primal and dual longest path length for maximum density st-planar graphs.

		p=0			**p=0.5**			**p=1**			$l \times l^*$		
n	$2n$	l	l^*	$l+l^*$	l	l^*	$l+l^*$	l	l^*	$l+l^*$	**p=0**	**p=0.5**	**p=1**
109	218	31	167	198	75	95	170	100	74	174	5177	7125	7400
222	444	42	374	416	105	216	321	151	129	280	15708	22680	19479
310	620	44	503	547	186	319	505	280	163	443	22132	59334	45640
436	872	100	524	624	248	412	660	397	178	575	52400	102176	70666
535	1070	98	785	883	240	534	774	402	293	695	76930	128160	117786
678	1356	80	1019	1099	382	449	831	625	195	820	81520	171518	121875
763	1526	144	1114	1258	385	780	1165	691	241	932	160416	300300	166531
863	1726	105	1286	1391	453	791	1244	767	270	1037	135030	358323	207090
998	1996	83	1419	1502	425	862	1287	846	340	1186	117777	366350	287640
1117	2234	109	1561	1670	551	902	1453	1013	208	1221	170149	497002	210704
1302	2604	134	2024	2158	704	1154	1858	1173	451	1624	271216	812416	529023
1410	2820	122	2120	2242	730	835	1565	1291	298	1589	258640	609550	384718
1501	3002	119	2203	2322	784	1073	1857	1403	224	1627	262157	841232	314272
1638	3276	110	2487	2597	833	1436	2269	1477	263	1740	273570	1196188	388451
1719	3438	131	2550	2681	856	1661	2517	1555	515	2070	334050	1421816	800825
1825	3650	180	2729	2909	886	1391	2277	1618	353	1971	491220	1232426	571154
1990	3980	208	2339	2547	1013	1581	2594	1773	400	2173	486512	1601553	709200
2089	4178	136	3095	3231	1002	1648	2650	1789	347	2136	420920	1651296	620783
2159	4318	142	3238	3380	930	1816	2746	1823	445	2268	459796	1688880	811235
2213	4426	162	3400	3562	1093	2082	3175	2008	551	2559	550800	2275626	1106408
2268	4536	148	3136	3284	952	1666	2618	1887	336	2223	464128	1586032	634032
2413	4826	154	3033	3187	971	1968	2939	1631	513	2144	467082	1910928	836703
4323	8646	356	5852	6208	2238	3589	5827	3957	841	4798	2083312	8032182	3327837
5102	10204	525	7155	7680	2597	4473	7070	4582	1139	5721	3756375	11616381	5218898

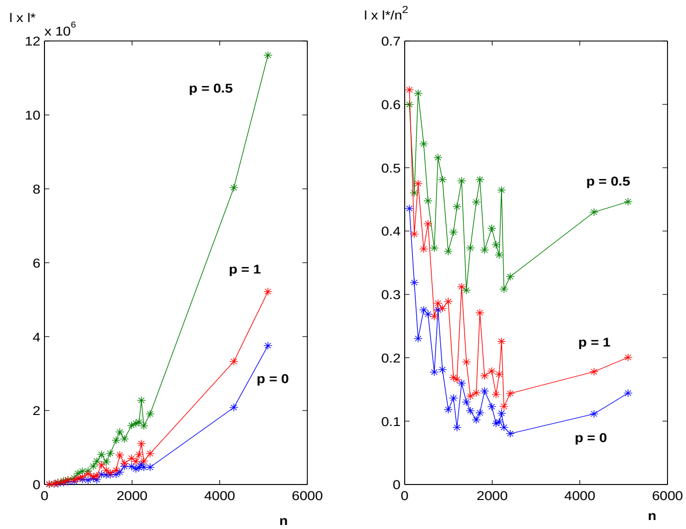


Figure 5.4: Absolute (left) and normalized (divided by n^2) (right) results for visibility representation area requirement for different values of the parameter p and maximum density planar graphs. The parameter $p = 0$ (low longest path *st*-oriented graphs) is clearly preferable.

From Table 5.8, it is clear that the primal and the dual longest path length are inversely proportional for various values of the parameter p. We have used the values $p = 0, 0.5, 1$, as the most representative ones. Additionally, it seems that for low density *st*-planar graphs the sum $l(t) + l^*(t)$ is no more that n (the number of the primal graph nodes), something that does not hold in general.

The last three columns of Table 5.8 show the product $l(t) \times l^*(t)$. This is actually the area that is needed in order to construct a visibility representation of the given graph using the algorithms proposed in [22]. The impact of the parameter p on the area is very evident. The savings in the area for different values of the parameter p is clear and actually for low density it is preferable to use the parameter $p = 1$. In Figure 5.3, we present a plot of the product $l(t) \times l^*(t)$ as a function of the size of the graph and the value of the parameter p.

In Table 5.9 and in Figure 5.4 the same results for triangulated planar graphs are presented. Note that for triangulated planar graphs the parameter value $p = 0$ is clearly preferable.

In Figure 5.5, we show 3 visibility representation frames of a 21-path planar graph produced with Pigale. The difference in the area is evident. Note that the visibility representation that uses the minimum *st*-orientation ($p = 0$) consumes the less area. Additionally, we present some other frames of parameterized visibility representations for both maximum and low density planar graphs (see Figures 5.6,5.7,5.8).

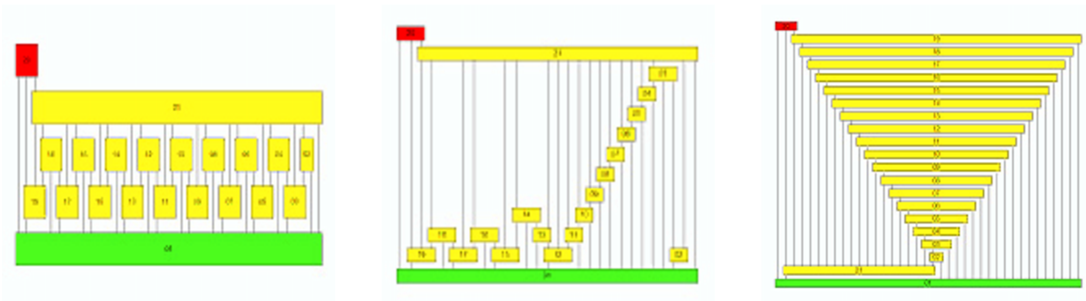


Figure 5.5: Visibility Representations of a 21-path planar graph for different st-orientations ($p = 0, 0.5, 1$).

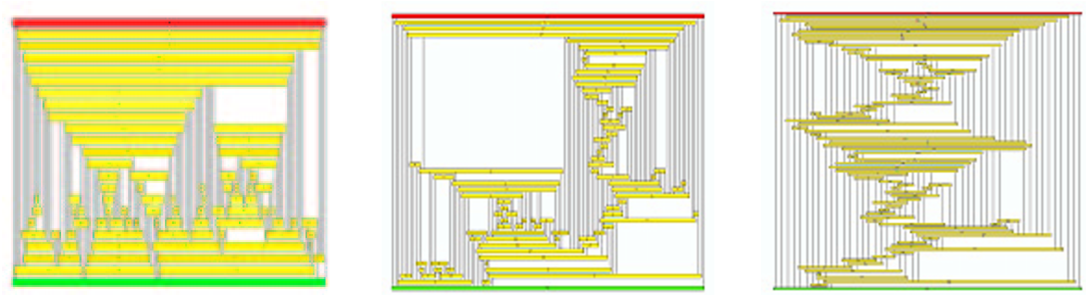


Figure 5.6: Visibility Representations of a 85-node triangulated planar graph for different st-orientations produced with PAR-STN(p) ($p = 0, 0.5, 1$).

Figure 5.7: Visibility Representations of a 100-node planar graph of density roughly equal to 1.5 for different st-orientations produced with PAR-STN(p) ($p = 0, 0.5, 1$).

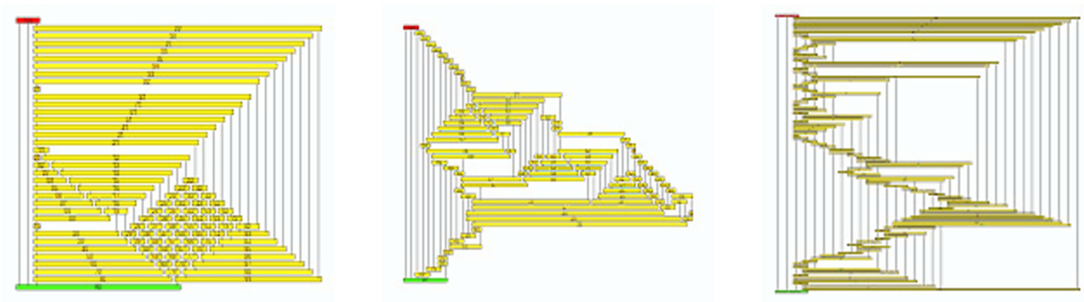


Figure 5.8: Visibility Representations of a 10x10 grid graph for different *st*-orientations produced with PAR-STN(p) ($p = 0, 0.25, 1$).

From Figures 5.6,5.7, we can see that the impact of the parameter p for different kind of graphs is obvious. Actually, for triangulated planar graphs (Figure 5.6) it is preferable to use the *st*-orientation computed with $p = 0$ whereas for low density planar graphs (Figure 5.7) it is preferable to use the orientation computed with $p = 1$, something that is indicated in the experimental results as well. Finally, in Figure 5.8 we present some visibility representations frames produced by *st*-orienting a grid graph. In this case, the importance of the parameter is clear. Using a parameterized *st*-orientation with $p = 0.25$ is preferable, as it produces a more *compact* drawing.

5.4 Orthogonal Drawings

In this section we present some applications of the parameterized *st*-orientations in orthogonal drawings.

Table 5.10: Area bounds for orthogonal drawings and different *st*-orientations.

n	**width** w			**height** h			$\frac{wh}{n^2}$		
n	$p = 0$	$p = 0.5$	$p = 1$	$p = 0$	$p = 0.5$	$p = 1$	$p = 0$	$p = 0.5$	$p = 1$
200	174	156	152	157	167	169	0.68	0.65	0.64
400	317	310	303	332	335	337	0.66	0.65	0.64
600	478	467	444	493	501	511	0.65	0.65	0.63
800	627	618	600	661	668	669	0.65	0.65	0.63
1000	790	742	728	819	848	850	0.65	0.63	0.62
1200	939	903	874	985	1009	1021	0.64	0.63	0.62
1400	1099	1052	1012	1146	1172	1191	0.64	0.63	0.61
1600	1240	1204	1166	1319	1346	1360	0.64	0.63	0.62
1800	1402	1363	1308	1479	1507	1525	0.64	0.63	0.62
2000	1527	1512	1444	1662	1673	1667	0.63	0.63	0.60

As we will see, the impact of longest path-parameterized *st*-orientations is not so big in the area of orthogonal drawings but it is worth mentioning it as a possible future research direction. In [18], an area-efficient algorithm to compute an orthogonal drawing is presented. Actually, this algorithm is applied to 4-degree graphs (the maximum degree is 4) and needs area at most $0.76n^2$. In the paper, it is stated that there is an impact of the st-numberings on the shape of the final orthogonal drawings. The algorithm uses an st-numbering is within a specific algorithm (pairing algorithm) that computes pairs of vertices. After the pairing algorithm is run on G we can compute the variables p_1 (number of column pairs), p_2 (number of unassigned degree-2 nodes) and p_3 (number of unassigned degree-3 nodes), k_2 (number of row pairs). Then by setting

$$k_1 = p_1 + p_2 + \frac{p_3}{2}$$

it is proved (in the paper) that the width of the drawing is $n + 1 - k_1$ and the height of the drawing is $n + 1 - k_2$. The pairing technique has been implemented (by using the parameterized st-numberings for $p = 0, 0.5, 1$) and we present some experimental results (see Table 5.10).

The impact of the different st-orientations is not very clear in orthogonal drawings, as indicated in Table 5.10. However, for the algorithm described in [18], where the area upper bound is roughly $0.76n^2$, we are able to produce st-numberings that produce drawings of area upper bound roughly equal to $0.68n^2$ or less.

5.5 Graph Coloring

In this section we present some experimental results concerning the use of MIN-STN in coloring graphs with the method described in the previous section. We have tested known benchmarks available at http://mat.gsia.cmu.edu/COLOR/instances.html.

From Table 5.11, we see that MIN-STN computes an almost optimal coloring for many of the benchmark graphs used. Actually, for the first 17 benchmark graphs G of Table 6, MIN-STN computes the chromatic number $\chi(G)$. For the last 7 benchmark graphs, MIN-STN computes a coloring equal to $\chi(G) + 1$. Note that all graphs used are of various densities. Additionally, they are constructed in a special way, which is fully described in the web address mentioned and which allows us to precompute their chromatic number.

In Table 5.12, we show the results for some benchmark graphs for which MIN-STN did not perform so well. Using MIN-STN to compute a good coloring of a graph G is not obviously the best approach to the graph coloring problem. It however reveals a cute application of parameterized st-orientations. The question that arises is whether MIN-STN can compute good colorings for other graphs as well. This is something that opens new research directions (maybe the development of new heuristics adjusted to the graph coloring problem in order to break the ties that appear in the choice of the next source during MIN-STN) and has to be tested experimentally.

Table 5.11: Benchmark graphs for which MIN-STN has computed an almost optimal coloring.

file name	**n**	**m**	**optimal coloring**	**MIN-STN (p=0) coloring**
myciel6.col	95	755	7	7
myciel5.col	47	236	6	6
myciel4.col	23	71	5	5
myciel3.col	11	20	4	4
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.5.col	186	3973	31	31
mulsol.i.4.col	185	3946	31	31
mulsol.i.3.col	184	3916	31	31
mulsol.i.2.col	188	3885	31	31
mulsol.i.1.col	197	3925	49	49
inithx.i.3.col	621	13969	31	31
inithx.i.1.col	864	18707	54	54
fpsol2.i.3.col	425	8688	30	30
fpsol2.i.1.col	496	11654	65	65
myciel7.col	191	2360	8	9
miles250.col	128	387	8	9
david.col	87	406	11	12
anna.col	138	493	11	12
zeroin.i.3.col	206	3540	30	31
zeroin.i.2.col	211	3541	30	31
inithx.i.2.col	645	13979	31	32

Table 5.12: Benchmark graphs for which MIN-STN has computed a relatively good coloring.

file name	**n**	**m**	**optimal coloring**	**MIN-STN (p=0) coloring**
queen8_12.col	96	1368	12	15
queen7_7.col	49	476	7	10
queen6_6.col	36	290	7	9
queen5_5.col	25	160	5	7
miles500.col	128	1170	20	23
homer.col	561	1629	13	15
fpsol2.i.2.col	451	8691	30	32

MIN-STN was used to compute coloring of other graphs (of known chromatic number) as well and computed a near optimal coloring.

Bibliography

[1] M. Mursalin Akon, S. Asaduzzaman, M.S. Rahman, and M. Matsumoto, *Proposal for st-routing*, Tellecommunication Systems **25(3-4)** (2004), 287–298.

[2] F. Annexstein and K. Berman, *Directional routing via generalized st-numberings*, SIAM J. Discrete Math. **13(2)** (2000), 268–279.

[3] G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis, *Annotated bibliography on graph drawing algorithms*, Computational Geometry: Theory and Applications **4** (1994), 235–282.

[4] ———, *Graph drawing: Algorithms for the visualization of graphs*, Prentice Hall, 1999.

[5] Ulrik Brandes, *Eager st-ordering*, LNCS ESA 2002, vol. 2461, 2002, pp. 247–256.

[6] N. Christophides, *Worst case analysis of a new heuristic for the traveling salesman problem*, Technical Report, CS-93-13, G.S.I.A., Carnegie Mellon University, Pittsburgh (1976).

[7] J. Ebert, *st-ordering the vertices of biconenected graphs*, Computing **30(1)** (1983), 19–33.

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

[9] H.D. Fraysseix, P.O. de Mendez, and P. Rosenstiehl, *Bipolar orientations revisited*, Discrete Applied Mathematics **56** (1995), 157–179.

[10] M. Garey and D. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, W. H. Freeman and Company, 1979.

[11] Michael T. Goodrich and Roberto Tamassia, *Data structures and algorithms in Java*, 2 ed., John Wiley & Sons, 2001.

[12] 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)** (2001), 723–760.

[13] J. Hopcroft and R. Tarjan, *Efficient algorithms for graph manipulation*, Comm. ACM **16** (1973), 372–378.

[14] Donald E. Knuth, *The art of computer programming — fundamental algorithms*, 3 ed., vol. 1, Addison-Wesley, 1997.

[15] A. Lempel, S. Even, and I. Cederbaum, *An algorithm for planarity testing of graphs*, P. Rosestiehl(ed.) Theory of Graphs: International Symposium July 1966, 1967, pp. 215–232.

[16] Y. Maon, B. Schieber, and U. Vishkin, *Parallel ear decomposition search (eds) and st-numbering in graphs*, Theoret. Comput. Sci **47** (1986), 277–298.

[17] S. Nakano, M.S. Rahman, and T. Nishizeki, *A linear-time algorithm for four-partitioning four-connected planar graphs*, Information Processing Letters **62** (1997), 315–322.

[18] A. Papakostas and I.G. Tollis, *Algorithms for area-efficient orthogonal drawings*, Computational Geometry: Theory and Applications **9** (1998), 83–110.

[19] P. Rosehnstiehl and R. Tarjan, *Rectilinear planar layout and bipolar orientation of planar graphs*, Discrete Comput. Geom. **1** (1986), 343–353.

[20] R.P. Stanley, *Acyclic orientations of graphs*, Discrete Math. **5** (1973), 171–178.

[21] K. Sugiyama, S. Tagawa, and M. Toda, *Methods for visual understanding of hierarchical systems*, IEEE Trans. on Systems, Man, and Cybernetics **SMC-11(2)** (1981), 109–125.

[22] R. Tamassia and I.G. Tollis, *A unified approach to visibility representations of planar graphs*, Disc. and Comp. Geom. **1** (1986), 321–341.

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

[24] ______, *Two streamlined depth-first search algorithms*, Fundamentae Informatica **9** (1986), 85–94.

[25] S. Vishwanathan, *An approximation algorithm for finding a long path in hamiltonian graphs*, Proc. of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, 2000, pp. 680–685.