IMPLEMENTATION OF A PLANARITY TESTING METHOD USING PQ-TREES

Alex William Cregten
Hannes Kristján Hannesson
T-404-LOKA
December 2017

Reykjavík University

Project report

Implementation of a planarity testing method using PQ-Trees

Alex William Cregten
Hannes Kristján Hannesson

December 2017

Abstract

The website GTea is introduced where two planarity testing algorithms have been implemented. Of these two algorithms, one is a brute-force method and the other a much faster PQ-Tree method introduced by K. S. Booth and G. S. Lueker. The two algorithms are discussed and running times compared in detail. A prerequisite algorithm to the PQ-Tree method is examined and implemented, which determines an st-numbering. The algorithm was introduced by S. Even and R. E. Tarjan. Front-end additions to GTea are shown which involve the manual modification and creation of graphs. A discussion on where this project has left GTea and the next steps forward are examined. The codebase of GTea can be found at the following link: `https://github.com/rostam/GTea/`.

Implementation of a planarity testing method using PQ-Trees

Alex William Cregten
Hannes Kristján Hannesson

desember 2017

Útdráttur

Vefsíðan GTea er kynnt, þar hafa verið útfærð tvö lagneta-prófana reiknirit. Af þessum reikniritum, þá nýtir annað sér jarðýtu aðferð, á meðan hitt er skilvirkara reiknirit sem nýtir sér gagnaskipanið PQ-Tré sem var kynnt af K. S. Booth og G. S. Lueker. Þessi reiknirit eru rædd og keyrslutímar þeirra eru bornir saman. Forsenda til að keyra PQ-Trája lagneta-prófana reikniritið, er að netið hafi st-tölusetningu, við ræðum útfærslu á reikniriti sem ákvarðar st-tölusetningu fyrir net, sem var kynnt af S. Even og R. E. Tarjan. Framenda lag-færingar á GTea sem leyfa handvirka breytingu og sköpun neta, og hugmyndir að viðbætum við GTea eru einnig ræddar. Kóðasafnið fyrir GTea er hægt að finna á eftirfarandi slóð: `https://github.com/rostam/GTea/`.

Acknowledgements

We want to thank Henning Arnór Úlfarsson and Christian Bean for guiding us through this project, Mohammad Ali Rostami for collaborating with us on GTea, Magnús Már Halldórsson for advice regarding choice of algorithm and time complexities, and Gylfi Þór Guðmundsson for advice regarding our references.

Contents

Acknowledgements **v**

Contents **vii**

List of Figures **ix**

List of Tables **xi**

List of Symbols **xiii**

1 Introduction **1**
1.1 Definitions and key terms 1
1.2 Background 3
1.3 Related Works 4

2 General improvements made to GTea **7**

3 Brute-force method **9**

4 The PQ-Tree method **11**
4.1 Overview 11
4.2 Biconnected components 11
4.3 st-numbering 12
4.4 PQ-Trees 14
4.5 Reductions 15
4.5.1 Templates 16
4.5.2 Bubble and Reduce 16
4.5.3 Planarity algorithm 18
4.5.4 Time complexity 19

5 Results **21**
5.1 Calculating time complexity 21
5.2 Doubling test 22
5.3 Comparison between algorithms 23

6 Discussion **27**
6.1 Summary 27
6.2 Conclusion 27
6.3 Where to go from here 28

Bibliography **31**

A Exhaustive list of templates used in PQ-Trees' reduction procedure **33**

B Running times **39**

List of Figures

1.1 Parallel edges between u and v . . . 2
1.2 Contracting the edge e between u and v . . . 3
1.3 Non-planar embedding of K_4 (left) and a planar embedding of K_4 (right) . . . 3

2.1 The GTea interface . . . 8

3.1 The two forbidden minors K_5 (left) and $K_{3,3}$ (right) . . . 9

4.1 Example PQ-Tree . . . 15
4.2 Two equivalent Q-nodes . . . 15
4.3 Full P-node and Q-node with $S = \{A, B\}$ . . . 16
4.4 Empty P-node and Q-node with $S = \{A, B\}$ . . . 16
4.5 Partial P-node and Q-node with $S = \{A, B\}$ . . . 16

5.1 log-log plot of the running time of the PQ-Tree method (without **`st-numbering`**) after optimization, compared to the number of edges, when running the algorithm on cycle graphs, green is the PQ-Tree, purple is $f(x) = x$, blue is $f(x) = x^2$, orange is $f(x) = x^3$, cyan is the trendline of the PQ-Tree . . . 22
5.2 Planarity testing on complete graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`** . . . 23
5.3 Planarity testing on complete graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`**, plot is scaled down . . . 23
5.4 Planarity testing on cycle graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`** (obscured) . . . 24
5.5 Planarity testing on cycle graphs, blue plot is brute-force, red is PQ-Tree before optimization (obscured), green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`** (obscured), plot is scaled down . . . 24
5.6 Planarity testing on gear graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization . . . 25
5.7 Planarity testing on gear graphs, blue plot is brute-force, red is PQ-Tree before optimization (obscured), green is PQ-Tree after optimization, plot is scaled down 25

A.1 Template L0 (Leaf) . . . 33
A.2 Template P0 . . . 33
A.3 Template P1 . . . 33
A.4 Template P2 . . . 34
A.5 Template P3 . . . 34

A.6 Template P4 35
A.7 Template P5 35
A.8 Template P6 36
A.9 Template Q0 36
A.10 Template Q1 36
A.11 Template Q2 37
A.12 Template Q3 38

List of Tables

B.1 Data collected from running the PQ-Tree method (without st-numbering, after optimization) on cycle graphs, used in Figure 5.1 39
B.2 Data collected from running the brute-force method on complete graphs 40
B.3 Data collected from running the PQ-Tree method on complete graphs (with st-numbering, before optimization) 43
B.4 Data collected from running the PQ-Tree method on complete graphs (with st-numbering, after optimization) 43
B.5 Data collected from running the PQ-Tree method on complete graphs (without st-numbering, after optimization) 44
B.6 Data collected from running the brute-force method on cycle graphs 44
B.7 Data collected from running the PQ-Tree method on cycle graphs (with st-numbering, before optimization) 45
B.8 Data collected from running the PQ-Tree method on cycle graphs (with st-numbering, after optimization) 46
B.9 Data collected from running the PQ-Tree method on cycle graphs (without st-numbering, after optimization) 47
B.10 Data collected from running the brute-force method on gear graphs 47
B.11 Data collected from running the PQ-Tree method on gear graphs (with st-numbering, before optimization) 48
B.12 Data collected from running the PQ-Tree method on gear graphs (with st-numbering, after optimization) 48

List of Symbols

Symbol	Description
G	Graph
V	Vertex set
E	Edge set
$V(G)$	Vertices of G
$E(G)$	Edges of G
$\mathcal{O}$	Asymptotic worst case time complexity

Chapter 1

Introduction

Computer-assisted mathematics is an important and growing topic in computer science. Today, computers can outperform any mathematician when computing properties of mathematical models that require a substantial number of calculations. Graph theory is a field of mathematics that benefits greatly from computational power. Graphs can be used in almost every field where data is concerned, such as computer science, medicine, economics, physics, and bioinformatics. Specifically, the algorithm this report is based on, concerning the graph characteristic known as planarity, can determine whether there exists a design such that no overlaps occur when designing systems such as roadways, microprocessors, and other similarly modeled systems. For example, a benefit of this for an architect could be to indicate whether the system they are planning to create can exist without collisions. On large systems, this could greatly reduce the workload by letting the architect know if time should be placed into designing it.

GTea is the codebase this report works on top of. It is a graph analysis tool developed by M. A. Rostami et al. The software was built using the existing codebase known as GraphTea, which was written by M. A. Rostami, A. Azadi, and M. Seydi [1]. The main difference between GTea and GraphTea is that GTea is to be deployed as a website, while GraphTea is stand-alone, client-side software.

The contribution we make to GTea is the functionality to automatically determine whether a biconnected graph is planar and the ability to manually draw and modify graphs. Our intentions have been to extend the codebase such that GTea can eventually be used by researchers, university students, and anyone in need of understanding particular characteristics of graphs, in particular, planarity.

1.1 Definitions and key terms

The following definitions and key terms are used throughout the report. The definitions and terms that pertain only to certain sections are defined in the relevant sections and not here.

Definition (Graph): A *graph* $G = (V, E)$ is a tuple consisting of a set of *vertices* V and *edges* E, where a vertex represents some object, and an edge represents a connection between two vertices.
The set of vertices of a graph G is denoted by $V(G)$, likewise, the set of edges is denoted by $E(G)$. An edge between vertices $u, v \in V(G)$ is denoted by (u, v). Let $N(v)$ be the *neighbors* of v, that is, the vertices that are reachable by traversing a single edge from v.

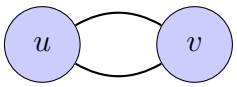


Figure 1.1: Parallel edges between u and v

Definition (Parallel edge): Let $u, v \in V(G)$. An edge $e = (u, v) \in E(G)$ is said to be a *parallel edge* if there exists another edge $e' = (u, v) \in E(G)$, an example can be seen in 1.1.

Definition (Loop): An edge is a *loop* if the edge is between the same vertex, i.e. $e = (v, v)$ where $v \in V(G)$, then e is considered a loop.

Definition (Null graph): A graph with 0 vertices and 0 edges is known as the *null graph*.

Definition (Simple graph): A graph G is *simple* if there are no parallel edges and no loops in $E(G)$.

Definition (Walk): A *walk*, is a non-empty, alternating sequence of vertices and edges [2, p. 12]. A walk is *closed* if the first and last vertex of the sequence are the same [2, p. 14].

Definition (Path): A *path*, is a walk where each vertex is distinct [2, p. 12].

Definition (Trail): A *trail*, is a walk where each edge is distinct [2, p. 12].

Definition (Face): A *face*, is an area of a graph G bounded by edges on the plane. An *infinite face* is the area of G with no such bound.

Definition (Wheel graph): A *wheel graph*, is a graph composed of a cycle, and a single vertex which has an edge to all vertices of the cycle.

Definition (Cycle graph): A *cycle graph*, is a trail, whose first vertex, and all but the last vertex, are distinct. [2, p. 14].

Definition (Gear graph): A *gear graph*, is composed of a wheel graph, where each edge (u, v) of the outer cycle in the wheel graph is replaced with two edges (u, w) and (w, v), where w is a new vertex [3, p. 58].

Definition (Complete graph): A *complete graph* is a graph where each vertex has an edge to every other vertex. A complete graph with n vertices is denoted by K_n.

Definition (Complete bipartite graph): A *complete bipartite graph*, is a graph where the set of vertices is the union of two disjoint sets of vertices A and B, such that every $a \in A$ has an edge to every $b \in B$, no $a \in A$ has an edge to $a' \in A$, and no $b \in B$ has an edge to $b' \in B$. A complete bipartite graph where $|A| = n$ and $|B| = m$, is denoted by $K_{n,m}$.

Definition (Biconnected graph): A *biconnected graph* is a connected graph that remains connected, when a single vertex and all of the edges connected to that vertex are removed.

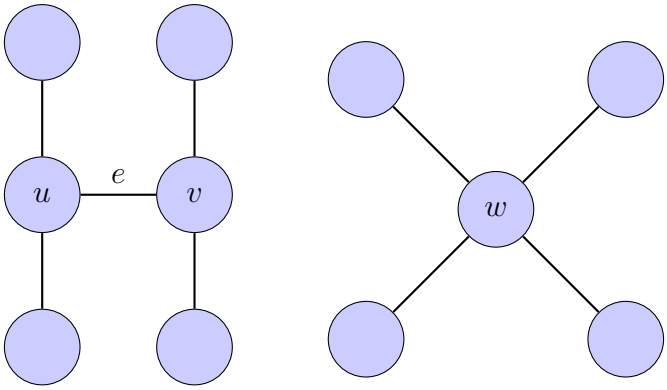


Figure 1.2: Contracting the edge e between u and v

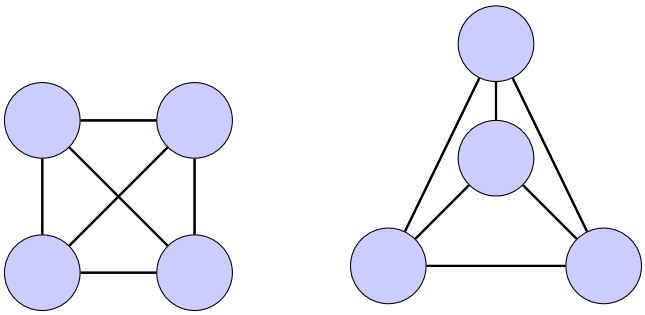


Figure 1.3: Non-planar embedding of K_4 (left) and a planar embedding of K_4 (right)

Definition (Simple edge contraction): "Let $G = (V, E)$ be a simple graph and e an edge (u, v) in E having end vertices u and v. Let w be a new vertex such that $w \notin V$"[4, p. 53]. We define a *simple edge contraction* to be the removal of the edge e, such that both end vertices u and v are replaced by w, where all edges that had u or v as one of its end vertices has w as its end vertex, Figure 1.2 demonstrates this.

Definition (Planarity): A graph is *planar*, if it is possible to draw the graph on a two-dimensional plane in a way such that no edges intersect. We call a graph that is drawn on a plane an *embedding*. If an embedding has no intersecting edges, we call it a *planar embedding*; conversely, if the embedding has intersecting edges, we call it a *non-planar embedding*. A particular graph can have multiple embeddings, as shown in Figure 1.3.

1.2 Background

Planar graphs have been a topic of study for many years. One of the most well-known theorems involving planar graphs was found in 1930 by the Polish mathematician C. Kuratowski. Kuratowski published a theorem defining what types of graphs are planar. His "Theorem on Planar Graphs" states that "A graph is planar, if and only if, it does not contain a subgraph homeomorphic to either K_5 or $K_{3,3}$" [5].

It was later, in 1937, that Klaus Wagner extended on Kuratowski's work by observing that the same applies to not only subgraphs but also graph minors [4, p. 210]. The combination

of these two theorems provides a clear definition of which graphs are planar and which are not. The difficulty, however, lies in finding whether a graph exhibits these traits.

Over time many algorithms to determine planarity, called planarity testing algorithms, emerged. The first linear time algorithm was made by J. Hopcroft and R. Tarjan [6] which built on the work of L. Auslander and S. V. Parter [7]. Their algorithm uses a depth-first search method with path addition.

There are a number of linear time planarity algorithms available but there exist none which are easy to implement. It was in 1976, K. S. Booth and G. S. Lueker provided a data structure known as a PQ-Tree which can be used for planarity testing [8]. This data structure provides a more easily implementable planarity tester, but in order to reach linear time the algorithm trades much of its ease of implementation.

This report will go into detail about the process of implementing this data-structure and the planarity testing algorithm in Java. For the complete source code, refer to the GitHub page of GTea[1].

1.3 Related Works

For software similar to GTea and GraphTea, the paper "Graphtea: Interactive graph self-teaching tool" written by M. A. Rostami, A. Azadi, and M. Seydi [1] lists several related software projects. The paper discusses how existing graph theory tools with good visualization have minimal features, while those with many features have minimal visualization and interactivity. GraphTea (and hence GTea) aims to add both good visualizations and many features by being general-purpose software "with the focus on education, that has rich editorial and visualization options" [1]. Rostami et al. describe five software competitors that have a focus on education, CARBI-Graph, Gato/CATBox, Tulip, EXPLAIN, and Health. They claim CARBRI-Graph and Gato/CATBox lack graph editing capabilities and Tulip requires Python knowledge which has a higher entry requirement for learners. They discuss that EXPLAIN and Health are good to teach with, but only deal with specific areas of Graph Theory.

Additionally, we found three competing online graph theory tools for identifying graph characteristics, Graphrel [9], Graphonline [10], and WolframAlpha [11]. Graphrel is an interactive site with good looking graphs but lacks many features such as planarity testing. Graphonline is interactive with more features, however, the number of features is still not great and it lacks planarity testing. WolframAlpha has many features and can test planarity but lacks interactivity, point and click graph creation and manipulation does not exist. GTea has a large number of features and a focus on great visualizations and graph interactivity. We felt GTea, while still not completely finished, would be a superior tool. It offered areas where we could add more functionality such as improved graph interaction, it also lacked a planarity testing algorithm; Both of which we wanted to work on.

Regarding planarity testing, there are a number of different approaches that achieve linear time. To name a few are J. M. Boyer and W. J. Myrvold's algorithm [12] which uses edge addition and a depth-first search, S. Wei-Kuan and H. Wen-Lian's algorithm [13] which

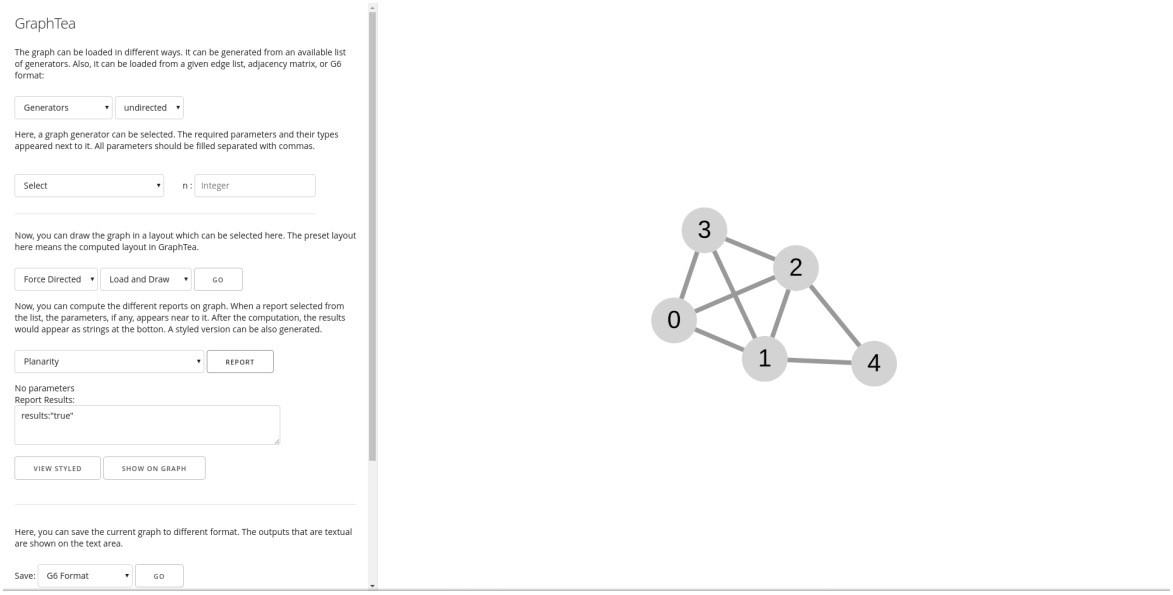
[1]The GTea project can be found at https://github.com/rostam/GTea/

uses a data-structure called a PC-Tree, and Hopcroft and Tarjan's which uses path addition [6].

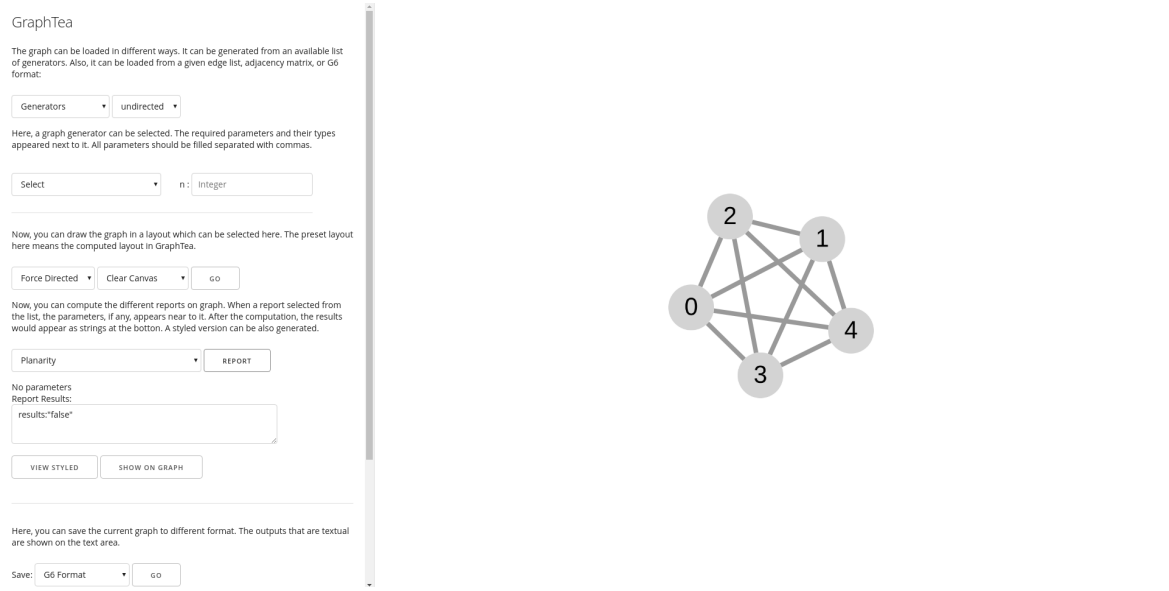
Chapter 2

General improvements made to GTea

The initial phase of this project was spent on getting GTea to a more user-friendly state. GTea did not have support for drawing arbitrary graphs on the canvas. Functionality was added to create vertices and draw edges manually as well as the deletion of edges and vertices. Directed and undirected edges were added, as well as functionality to clear the canvas. Lastly, we implemented sessions so multiple users would not share data. The interface can be seen in Figure 2.1.



(a) Planar graph on the canvas in GTea



(b) Non-planar graph on the canvas in GTea

Figure 2.1: The GTea interface

Chapter 3

Brute-force method

We first implemented a brute-force algorithm based on Wagner's theorem. Our intentions were to use this algorithm as a base to test against a faster PQ-Tree algorithm discussed in Chapter 4.

Theorem (Wagner): A graph G is planar, if and only if, it does not contain a minor K_5 or $K_{3,3}$ [4, p. 210].

The graphs K_5 and $K_{3,3}$ are known as *forbidden minors*, since their presence in a graph makes it non-planar.

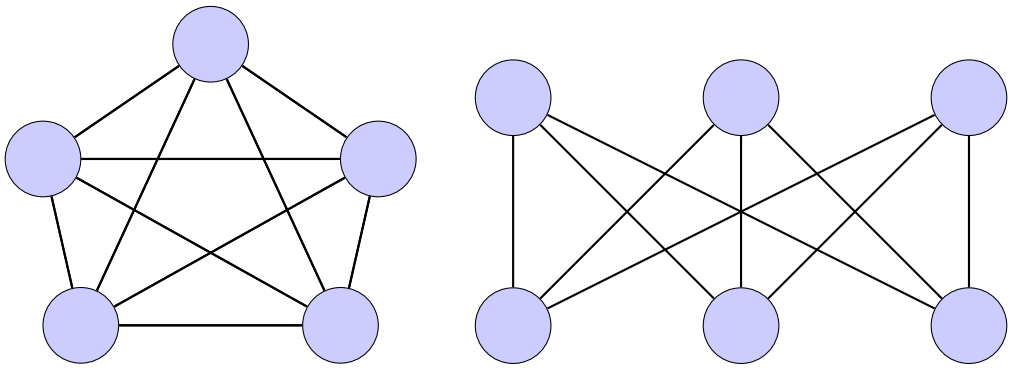


Figure 3.1: The two forbidden minors K_5 (left) and $K_{3,3}$ (right)

The algorithm **`brute-force`** utilizes Wagner's theorem by contracting edges and removing vertices until a K_5 or $K_{3,3}$ minor is found. This algorithm is not advised for anyone to implement for anything other than small graphs as it has an $\mathcal{O}((V+E)^{(V+E)})$ running time. The reasoning for this is that the algorithm tries all possible minors which result from removing a vertex or contracting an edge. This results in $V+E$ choices of vertices for removal and edges for contraction. The maximum number of vertex deletions and edge contractions we can apply before reaching a graph with 0 vertices and 0 edges on a single graph is $V+E$, thus we have $\mathcal{O}((V+E)^{(V+E))}$ operations. Its purpose is to test the more complex PQ-Tree algorithm as described in [8] for correctness.

Algorithm 1 `brute-force`

Input: Graph G
Output: Boolean indicating planarity

```
if G is K_5 or K_{3,3} then
    return false
end if
for each edge e ∈ E(G) do
    let G' be G after a simple edge contraction on e
    if not brute-force(G') then
        return false
    end if
end for
for each vertex v ∈ V(G) do
    Let G'' := G(V − {v}, E)
    if not brute-force(G'') then
        return false
    end if
end for
return true
```

Chapter 4

The PQ-Tree method

We introduce and outline the functionality of the PQ-Tree data-structure and the linear time planarity testing algorithm as described in [8]. A caveat of this method is that it only handles graphs which are biconnected. Known algorithms that produce biconnected components exist that require $\mathcal{O}(n + e)$ steps for graphs having n vertices and e edges [8, p. 373] so this would not affect overall running time, if implemented correctly.

4.1 Overview

There are three steps involved in order to construct a complete planarity tester using Booth and Lueker's PQ-Tree method. First, a graph must be biconnected. Any graph can be made biconnected by splitting it into biconnected components. Next, each biconnected component must be given a valid vertex numbering known as an st-numbering. Lastly, each st-numbered component is placed into the PQ-Tree algorithm and tested for planarity. The following sections discuss each procedure. The proof for why this works is discussed in K. S. Booth and G. S. Lueker's paper [8].

4.2 Biconnected components

We can extend the PQ-Tree planarity testing algorithm to work on any graph. We can split the graph into its biconnected components and run **`planar`** on them individually, and combine the results to detemine if the graph as a whole is planar. We know this is possible by Lemmas 1 and 2.

Lemma 1: All graphs (apart from the null graph, and a graph containing 1 vertex) contains 1 or more biconnected components.

Lemma 2: A graph is planar if, and only if, all of its biconnected components are planar [8].

As stated earlier, algorithms exist that require $\mathcal{O}(n + e)$ steps for graphs having n vertices and e edges. Because we did not implement this, and only focus on naturally biconnected graphs, we will leave this section as an exercise for future developers to research more thoroughly.

4.3 st-numbering

A valid st-numbering of G is a vertex numbering such that 1 and n are adjacent and, for any vertex $1 < j < n$ there are vertices numbered i and k such that $i < j < K$ [8, p. 373].

Lemma 3: If G is biconnected there exists a valid st-numbering for its vertices.

A linear time st-numbering process by S. Even and R. E. Tarjan is followed. There are three parts to this algorithm, **`pre-order`**, **`pathfinder`**, and **`st-numbering`**. Each part contributes to the entire st-numbering process. The parts are described below and are a slightly modified version of the one described by Even and Tarjan in [14, pp. 339–344].

The first step in the st-numbering process is to generate a vertex numbering which corresponds to the order in which vertices are visited in a depth-first search.

Let $pre(v)$ be the pre-order numbering of vertex $v \in V(G)$.

Algorithm 2 `pre-order`

Input: Tree T, vertex v to start at
Output: None

give $pre(v)$ a value such that $pre(v) > max(\{pre(u) \mid u \in V(G) \wedge v \neq u\} \cup \{0\})$
for each $w \in N(v)$ **do**
    `pre-order(`w**`)`**
end for

Once a pre-order numbering has been constructed, the algorithm **`st-numbering`** is run which calls a function named **`pathfinder`**. The function **`pathfinder`** identifies what types of paths to return from the input graph G with specific vertices passed in by **`st-numbering`**. The end goal is to find a unique path within G. An important part of this function is to never process an edge that has already been processed in any other iteration of this function and to never process a vertex in which all of its edges have been processed.

The following definitions are used in the **`pathfinder`** pseudo-code detailed further on.

Definition (backedge): Let the backedge of a DFS traversal be an edge that connects the two vertices u and v, such that after adding the backedge, there exists a cycle from u to u via v.

Definition (L(v)): Let $L(v) = \{\min(\{v\} \cup \{u \mid$
$\exists w$ such that there is a path from v to w and w has a backedge to $u\})\}$[14, p. 341].

Definition (*$\rightarrow$): Let $v* \rightarrow w$ denote the edge where v is an ancestor to w (i.e. the pre-order numbering of v is less than the pre-order numbering of w).

Algorithm 3 `pathfinder`

Input: Vertex v
Output: A path $(v_1, v_2), \ldots, (v_{k-1}, v_k)$

if there exists a cycle edge (v, w) with $w* \rightarrow v$ **then**
  mark (v, w) old
  let path be (v, w)
  return path
end if
if there is a new tree edge $v \rightarrow w$ **then**
  mark (v, w) old
  initialize path to be (v, w)
  while w is new **do**
    find a (new) edge (w, x) with $x = L(w)$ or $L(x) = L(w)$
    mark w and (w, x) old
    add (w, x) to path
    $w := x$
  end while
  return path
end if
if there is a new cycle edge (v, w) with $v* \rightarrow w$ **then**
  mark (v, w) old
  initialize path to be (v, w)
  while do
    mark w and (w, x) old
    add (w, x) to path
    $w := x$
  end while
  return path
else
  return empty path
end if

The algorithm **`st-numbering`** computes the st-numbering of each vertex. A stack is initialized with two arbitrary adjacent vertices $s, t \in G(V)$ such that s is placed on top of t. While the stack is not empty a vertex v is popped off and **`pathfinder`** returns a path from v to some vertex w. If no path was found v is given a unique st-number, otherwise, that path is placed onto the stack. At no point will the same vertex be in the stack more than once. This ensures that each st-numbered vertex is final.

Algorithm 4 `st-numbering`

Input: Graph G
Output: A mapping of each vertex $v \in G$ to its new st-numbering

```
mark s, t and (s, t) old, and all other vertices and edges new
initialize stack to contain s on top of t
i := 0
while stack is not empty do
    let v be the top vertex of stack
    delete v from stack
    let (v_1, v_2), ..., (v_{k-1}, v_k) be path found by pathfinder(v)
    if path not null then
        add v_{k-1}, ... ,v_1 (v_1 = v on top ) to stack
    else
        map v to i := i + 1
    end if
end while
return mapping
```

4.4 PQ-Trees

A PQ-Tree is a data-structure which holds the possible permutations of a set U, known as the universal set, under certain constraints. Its construction involves three types of nodes, P-nodes, Q-nodes, and leaf nodes. The P-nodes and Q-nodes hold certain properties that allow the tree to contain a set of permutations of the leaves.

Definition (P-node): A P-node is a node in a PQ-Tree, whose children can be arbitrarily ordered.

Two P-nodes are equivalent if their sets of children are identical, this is because they have no imposed order.

Definition (Q-node): A Q-Node is a node in a PQ-Tree, whose children can be ordered in a left-to-right, or right-to-left manner.

Q-nodes have an imposed order, so the order of children must be identical or symmetric for two Q-nodes to be equivalent. They are modeled as rectangles and seen in Figure 4.2.

Definition (Leaf node): A leaf node is a node in a PQ-Tree that translates to an element in the set U.

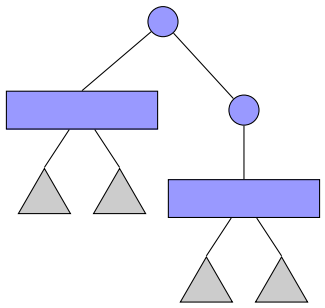


Figure 4.1: Example PQ-Tree

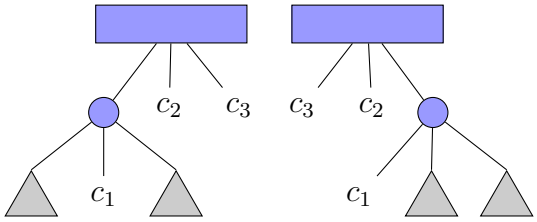


Figure 4.2: Two equivalent Q-nodes

Leaves can not have any children, thus have no ordering.

Definition (PQ-Tree): A PQ-Tree is an n-ary tree, where each node is either a P-node, a Q-node, or a leaf.

Definition (S): The set $S \subset U$ contains elements which must occur as a consecutive sequence in the related PQ-Tree and is known as a constraint.

4.5 Reductions

Given a PQ-Tree T and a constraint S, we want to **`reduce`** T such that it does not violate the constraint S. The reduction is arguably the most important part of the algorithm since it is reconstructing the tree. If no reconstruction can be found, then there does not exist a permutation where S appears consecutively in U and $null$ is returned. K. S. Booth and G. S. Lueker provide a linear-time algorithm for PQ-Tree reductions in [8];

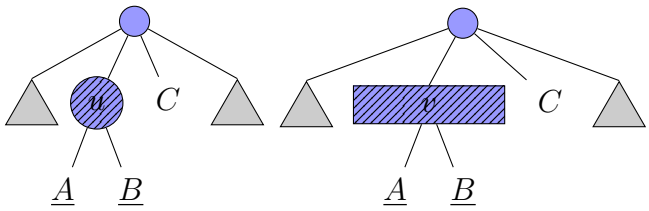
Definition (Full node): A node X is *full* if all of its descendants are in S [8, p. 343].

Full nodes are shown as shaded as seen in Figure 4.3.

Definition (Empty node): A node X is *empty* if none of its descendants are in S [8, p. 343].

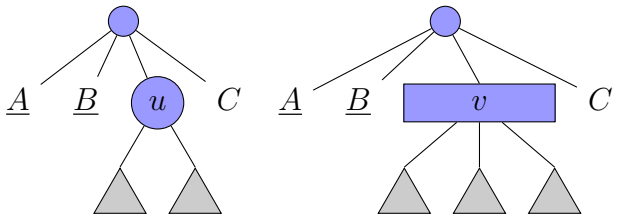
Empty nodes are shown as clear as seen in Figure 4.4.

Definition (Partial node): A node X is *partial* if some but not all of the descendants are in S [8, p. 343].



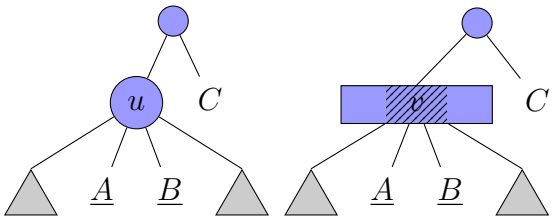
(a) A full P-node, u (b) A full Q-node, v

Figure 4.3: Full P-node and Q-node with $S = \{A, B\}$



(a) An empty P-node, u (b) An empty Q-node, v

Figure 4.4: Empty P-node and Q-node with $S = \{A, B\}$



(a) A partial P-node, u (b) A partial Q-node, v

Figure 4.5: Partial P-node and Q-node with $S = \{A, B\}$

Partial nodes are shown as shaded in Q-nodes and clear in P-nodes Figure 4.5.

Definition (Pertinent): A node X is *pertinent* if it is *partial* or *full* [8, p. 343].

4.5.1 Templates

A template consists of two components, the template *pattern*, and the template *replacement*. The template patterns are essentially a structure of a PQ-Tree, which has a corresponding replacement. Replacements are instructions on how to restructure the PQ-Tree (or a subtree of it). There are 12 templates in total, which can be found in Appendix A.

4.5.2 Bubble and Reduce

Booth and Lueker's algorithm starts off by marking each (non-leaf) node in the PQ-Tree with the number of pertinent children, in a bottom-up manner. This results in each node having information about how many of its children are pertinent. This procedure is called

bubble and can be found in [8, pp. 358-359]. The main purpose of **bubble**, is to speed up the reduction process, as we only need to process the pertinent children and their parents in the tree.

Using the information **bubble** writes to the interior nodes, we can efficiently process each element of S in the tree during the procedure **reduce**. The elements of S we will consider as nodes in the PQ-Tree.

As each node in S is processed from the bottom up, the ancestors of those nodes are processed as well. During **reduce**, templates are matched that modify the tree such that every node in S is sequential with the same parent. If there is no such template (or combination of templates) that allow the tree to be modified in such a way, then the input does not contain a permissible permutation with the constraint of S. In other words, there exists no permutation of the leaf nodes in T such that S appears sequentially within that permutation.

In the case of planarity, the nodes in U, and hence S, map to edges. If no template can be applied, then there exists no permissible ordering of these edges such that each edge is adjacent in the plane and so at least two edges will overlap.

Algorithm 5 reduce

Input: PQ-Tree T
Output: Constraint S

initialize Q to be an empty queue
for each leaf $X \in U$ **do**
    place X onto Q
end for
while Q is not empty **do**
    remove X from the front of Q
    if some template applies to X **then**
        substitute the replacement for the pattern in T
    else
        let T be the null tree
        exit from while loop
    end if
    if $S \subset \{Y : X$ is an ancestor of Y **then**
        exit from while loop
    end if
    if every sibling of X has been matched **then**
        place the parent of X onto Q
    end if
end while
return T

4.5.3 Planarity algorithm

The algorithm checks that each edge placed into S is able to be modeled consecutively in the PQ-Tree. A consecutive modeling translates to the edges being able to be drawn consecutively in the plane, which therefore maps to a planar embedding; e.g. the edges $(2,1),(3,1),(4,1) \in S$ are consecutive such that $(2,1)$ is next to $(3,1)$ is next to $(4,1)$, then $(2,1)$ shares a face with $(3,1)$ and $(3,1)$ shares a face with $(4,1)$.

One of the prerequisites, in order to run **`planar`**, is that the graph has an st-numbering. The purpose of a graph having an st-numbering is to ensure that the set S and S' are never empty, when running **`planar`**. The sets S and S' are created within each iteration of a value j. The edges whose lower numbered vertex is j are placed into S and the edges whose higher numbered vertex is j are place into S'. It is this way that the graph is processed and embeddings are created.

Before the iterations start, the edges connected to vertex 0 can be drawn as a planar embedding, since there is only one vertex, the graph is planar at this point. The tree T is initialized such that any permutations of those edges are permissible. The algorithm **`planar`** iterates through the graph edges and places them into the set S', it then applies reductions on S where necessary by calling **`bubble`** and **`reduce`**. Once **`reduce`** has finished running, S' replaces S nodes in the PQ-Tree T.

The algorithm **`planar`** is a paraphrasing of an algorithm found in [8, pp. 373–374].

Algorithm 6 `planar`

Input: Graph G
Output: Boolean indicating planarity

Let U be the set of edges whose lower-numbered vertex is 1
$T := T(U, U)$
for $j := 2, 3, \ldots, n-1$ **do**
    Let S be the set of edges whose higher-numbered vertex is j
    `bubble`(T, S)
    `reduce`(T, S)
    if T happens to be the empty tree **then**
        return false
    end if
    Let S' be the set of edges whose lower-numbered vertex is j
    if the root of the pertinent subtree is a Q-node **then**
        replace the full children of the pertinent root with a P-node, containing all of S' as children
    else
        replace the pertinent subtree with a P-node containing all of S' as children
    end if
    $U := (U - S) \cup S'$
end for

4.5.4 Time complexity

From our analysis, we have determined that the running time is proportional to at least $V + E$, since **`planar`** iterates through the vertices, and for each vertex iterates over the neighbors of that vertex.
We believe that the time complexity is not linear due to the fact that we did not implement the suggestions Booth and Lueker suggest in [8, pp. 351–356], which involves taking advantage of the fact that we can skip processing some children of Q-nodes, by marking them as blocked. For more conclusive results, see 6.2.

Chapter 5

Results

All data in this section is gathered by running the algorithms on a notebook computer with an Intel 5300U processor and 8 Gigabytes of memory. Note that the running time of the PQ-Tree planarity testing algorithm is determined by the number of vertices and edges, however, we chose to plot the times with respect to the edges only, due to clarity. One interesting note about this data, is that the brute-force method handled complete graphs better than the other types of graphs.

We improved on the performance of the PQ-Tree planarity testing algorithm by constructing our own function to determine if the graph was biconnected, as the class "KConnected.java" performed poorly. Thus, the version before this optimization we call *before optimization*, and the version after the optimization we call *after optimization*.

5.1 Calculating time complexity

We ran the optimized PQ-Tree planarity algorithm on a series of inputs, such that each term has double the number of edges of the previous term, and plotted the size of the input against the running time for that specific input. Note that for the log-log plots, we omitted the process of computing the st-numbering, as the numbering of the vertices does not matter for cycle and complete graphs (since complete graphs are naturally st-numbered, and cycle graphs are st-numbered in GTea by default). This was done to gather data for larger graphs, as the st-numbering process would cause a stack overflow at more than a certain number of vertices and edges (depends on stack size specified in the Java Virtual Machine). We did not do a doubling test on complete graphs, as doubling the number of edges would not always give us a graph with an integer number of vertices (since the number of edges in K_n is $\frac{n(n-1)}{2}$). Some graphs have the property that any numbering given to the vertices happens to already be an st-numbering, thus there is no need to run the st-numbering algorithm for these kinds of graphs; unfortunately, our algorithm does not determine this. We will compare running the algorithm with and without the st-numbering algorithm, on those graphs which meet these criteria (cycle and complete). Note that for our measurements of the time complexity, we only counted running times for when the running times were above 0.3 seconds for PQ-Tree when doing the doubling test, as the running times of smaller inputs tend to be less reliable.

5.2 Doubling test

The data for Figure 5.1 can be found in Appendix B.

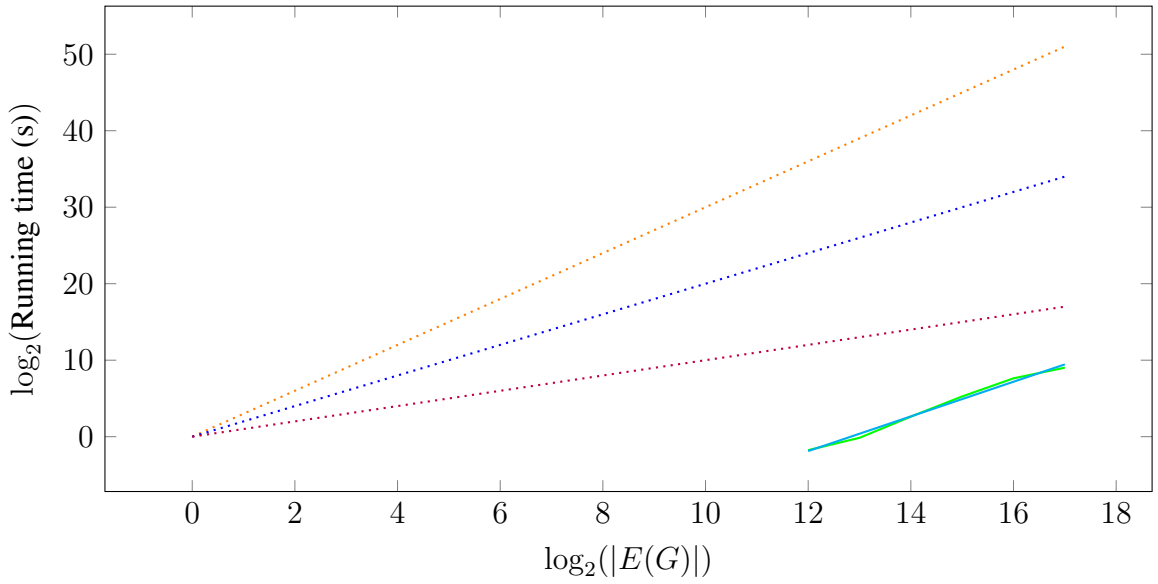


Figure 5.1: log-log plot of the running time of the PQ-Tree method (without **`st-numbering`**) after optimization, compared to the number of edges, when running the algorithm on cycle graphs, green is the PQ-Tree, purple is $f(x) = x$, blue is $f(x) = x^2$, orange is $f(x) = x^3$, cyan is the trendline of the PQ-Tree

5.3 Comparison between algorithms

The figures in this section demonstrate the difference of time complexities between the algorithms, with respect to the number of edges.

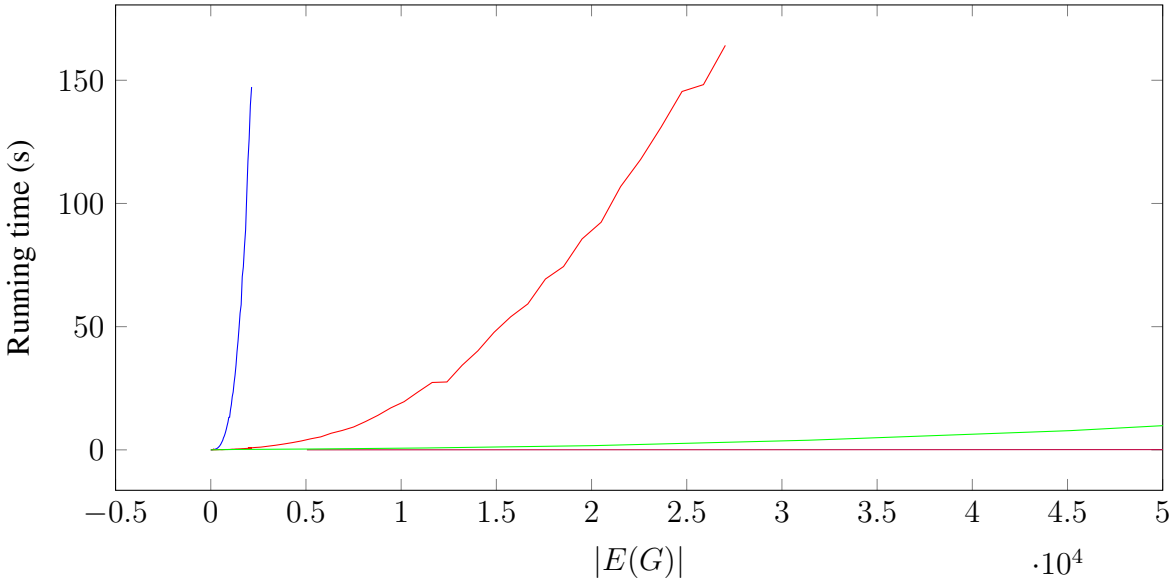


Figure 5.2: Planarity testing on complete graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`**

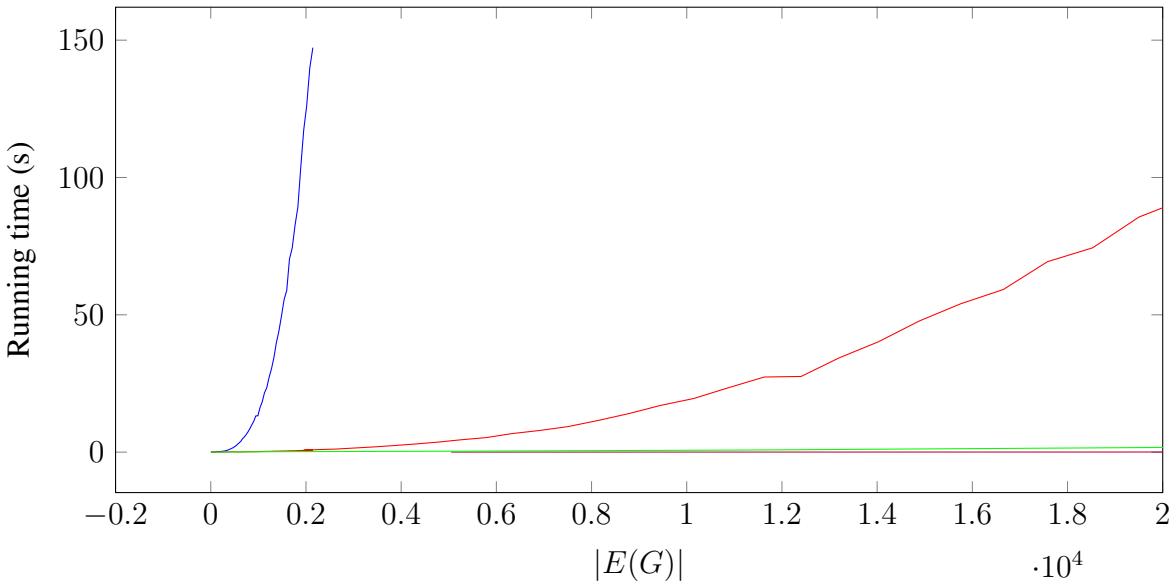


Figure 5.3: Planarity testing on complete graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`**, plot is scaled down

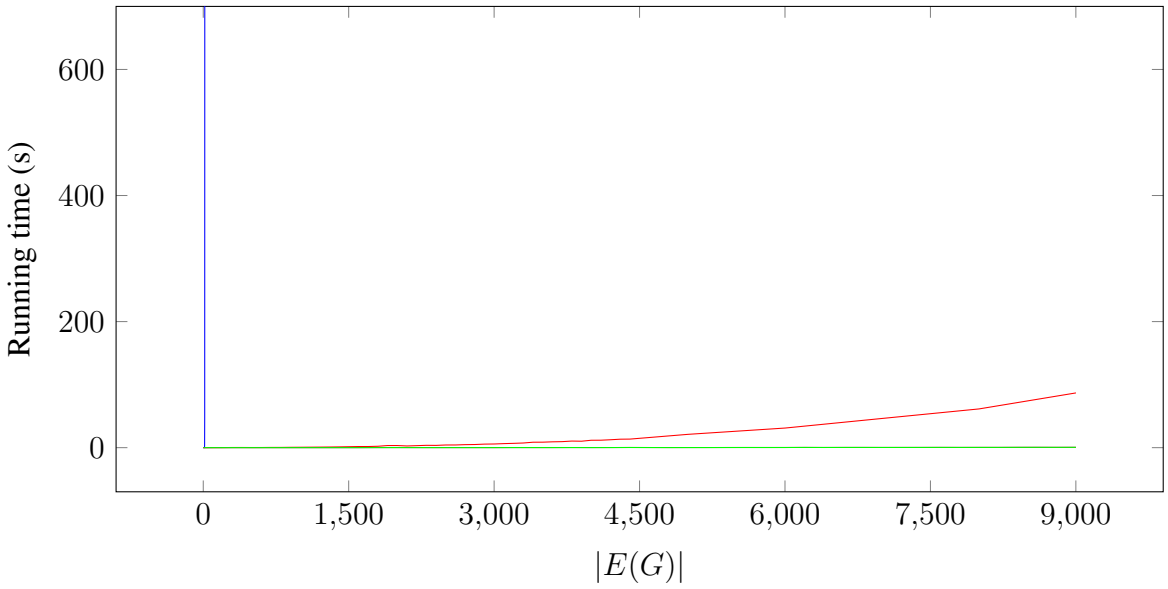


Figure 5.4: Planarity testing on cycle graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`** (obscured)

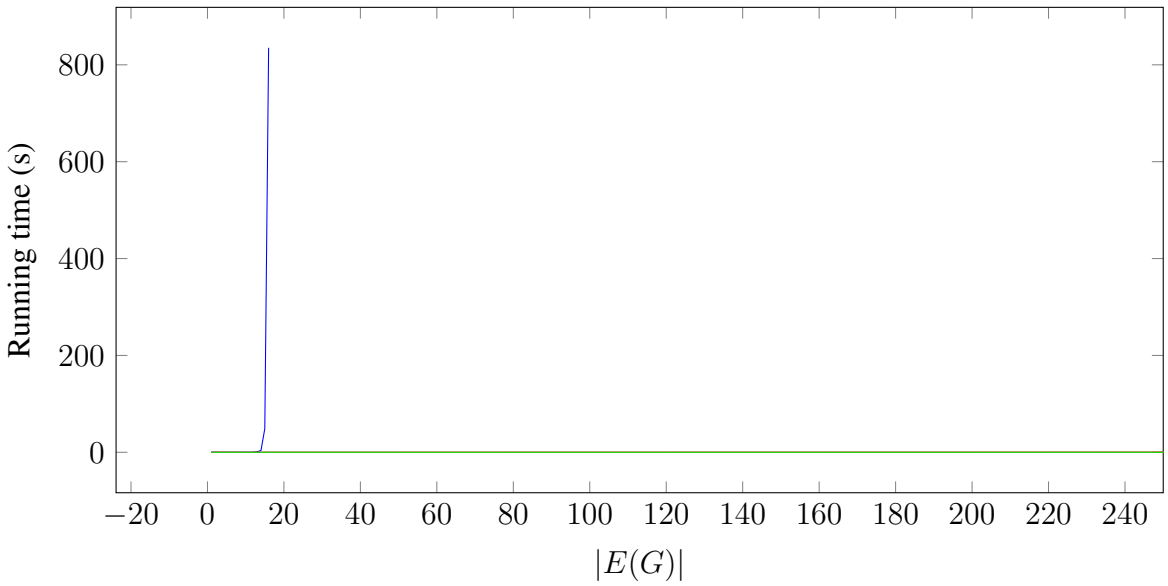


Figure 5.5: Planarity testing on cycle graphs, blue plot is brute-force, red is PQ-Tree before optimization (obscured), green is PQ-Tree after optimization, purple is PQ-Tree after optimization without **`st-numbering`** (obscured), plot is scaled down

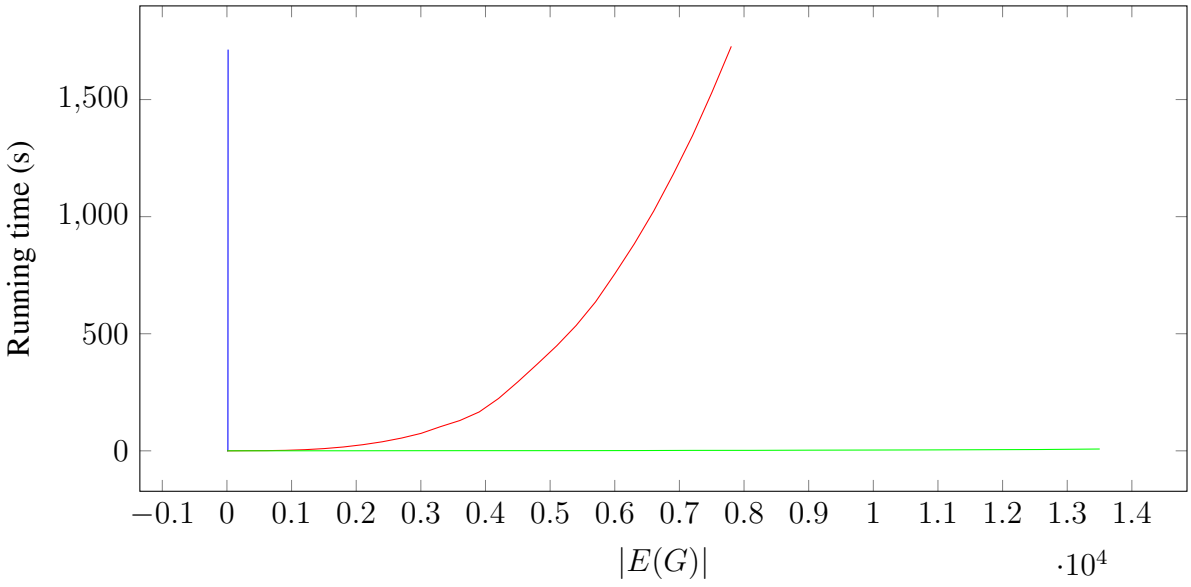


Figure 5.6: Planarity testing on gear graphs, blue plot is brute-force, red is PQ-Tree before optimization, green is PQ-Tree after optimization

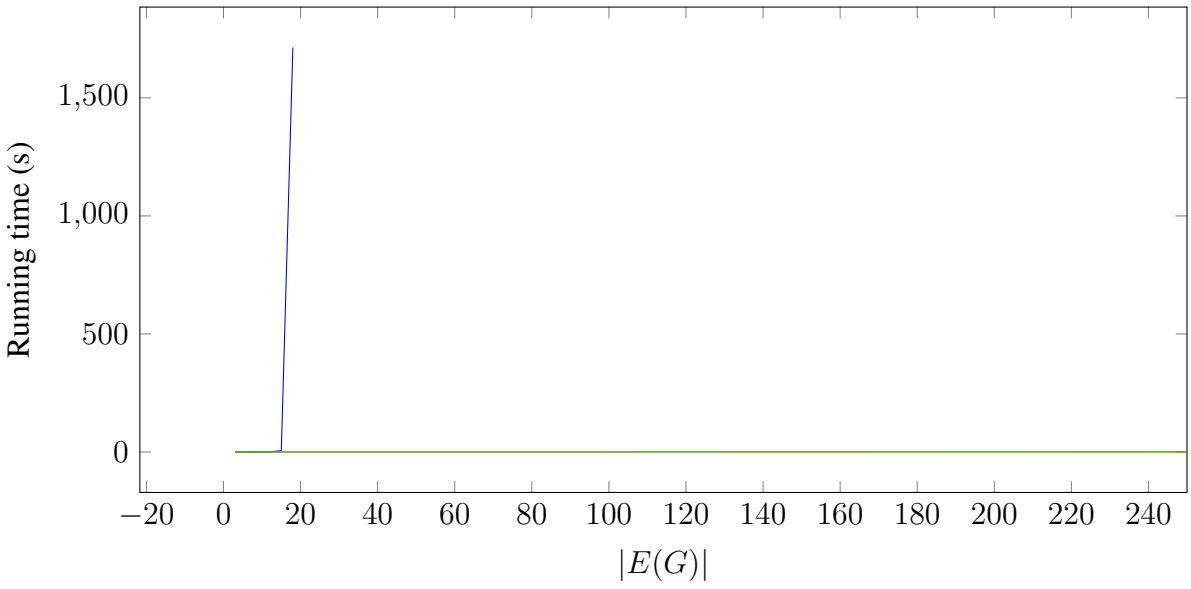


Figure 5.7: Planarity testing on gear graphs, blue plot is brute-force, red is PQ-Tree before optimization (obscured), green is PQ-Tree after optimization, plot is scaled down

Chapter 6

Discussion

6.1 Summary

We discussed and implemented two planarity testing algorithms. First, a naive brute-force planarity tester based on Wagner's theorem and second, a faster one involving PQ-Trees by K. S. Booth and G. S. Lueker. The brute-force version was clearly too slow for any practical use other than initially testing the PQ-Tree algorithm for correctness on small graphs. The PQ-Tree algorithm was an easy winner in terms of speed, however, it also has a difficult implementation along with two prerequisite algorithms needed.

We added some front-end functionality to GTea that enabled the creation and modification of manual graphs by the end user, added sessions to handle multiple users, and provided the ability to update a graph to be either directed or undirected.

6.2 Conclusion

We came to the conclusion that the optimized PQ-Tree planarity testing algorithm (without st-numbering) runs in time proportional to $((|V| + |E|)^{2.12})$ on cycle graphs, which gives a good estimate for graphs in general. The reason why we are using $V + E$ instead of E, is that both $|V|$ and $|E|$ affect the running time, however, when plotting the data in 5, we decided to go with a simpler approach and only plot with respect to $|E|$. This conclusion was based on the data presented in Chapter 5. This deviates from the linear time version we tried to implement. A likely cause of this is that our version of **`bubble`** traverses all of the Q-node children when obtaining a pertinent child count. It is explained by Booth and Lueker that in order to reach linear time **`bubble`** should only process one pertinent child per Q-node [8, pp. 351-355].

The **`st-numbering`** algorithm cannot be properly tested for its time complexity. This is because a stack overflow occurs that only allows the algorithm to handle a maximum number of vertices of a graph, and the number happens to be lower than the size of the largest test input; note that testing a smaller input is not really an option since that would not give reliable running times; however, we estimate that it is not quite linear from the data shown in Chapter 5, which deviates from what Even and Tarjan's paper states [14].

The point at which the program suffers from a stack overflow is determined by the number of vertices rather than the number of edges; since we mostly cached properties relating to vertices. We observed that **`planar`** with **`st-numbering`** can handle a cycle graph of

9001 edges and 9001 vertices but overflows for a cycle graph of 9201 edges and 9201 vertices. An improvement for space would be to not cache as much data as we are currently. The reason so much was cached was to improve performance issues. We felt the average website needs to be quick in order for users to be satisfied and so we opted for speed.

Even though we aimed for a linear timed implementation of a planarity tester, it is important to note that GTea users may not have a need for such speed. Our implementation can handle a 9001 edge cycle graph in 0.55 seconds, and a 5403 edge gear graph in 0.8 seconds with st-numbering enabled, which we feel is fast enough.

6.3 Where to go from here

The PQ-Tree planarity testing algorithm and additions to GTea have been quite involved, however, what we have worked on can be improved in various ways. The following discusses functionality that would improve the PQ-Tree algorithm in terms of speed and functionality, add another use-case for the algorithm, and include more uses for the PQ-Tree data-structure.

A prerequisite to running **`st-numbering`** is that the graph must be biconnected. By adding a feature that breaks a graph up into biconnected components the planarity tester would achieve correct testing for all input graphs. Algorithms exist that can achieve this in linear time [8, p. 373].

In terms of speed, the time complexity could be improved to linear by limiting the linear Q-node searching when traversing the children in **`bubble`**. This process is described in [8, pp. 351-355].

GTea is dedicated to offering good visualizations and animations to end users. The planarity testing algorithm currently only provides a true or false output. However, the algorithm travels through the graph and models planar embeddings in the PQ-Tree as it goes. Each new vertex traveled to within each iteration of **`planar`** models a subgraph which contains all of the currently traversed edges and vertices. Therefore, it is possible to draw an embedding as the algorithm progresses, drawing the embedding would be a nice addition to the website.

It is important to note that the PQ-Tree data structure is not confined to only planarity testing. It can be used in many more linear time algorithms. A few include identifying interval graphs, and whether a graph has the consecutive one's property [8]. Implementing these algorithms should be more manageable, since the PQ-Tree data-structure has already been implemented.

Bibliography

[1] M. A. Rostami, A. Azadi, and M. Seydi, "Graphtea: Interactive graph self-teaching tool", in *Proc. 2014 Int. Conf. Edu. & Educat. Technol. II*, 2014, pp. 48–52.

[2] J. A. Bondy, U. S. R. Murty, *et al.*, *Graph theory with applications*. 1976, vol. 290.

[3] M. Rahim, M. Farooq, M. Ali, and S. Jan, "Multi-level distance labelings for generalized gear graph", *International Journal of Mathematics and soft computing*, vol. 2, no. 1, 2012.

[4] G. Agnarsson and R. Greenlaw, *Graph Theory: Modeling, Applications, and Algorithms*. 2007.

[5] J. W. Kennedy, L. V. Quintas, and M. M. Sysło, "The theorem on planar graphs", *Historia Mathematica*, vol. 12, no. 4, pp. 356–368, 1985. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/031508608590045X`.

[6] J. Hopcroft and R. Tarjan, "Efficient Planarity Testing", *J. ACM*, vol. 21, no. 4, pp. 549–568, Oct. 1974.

[7] L. Auslander and S. V. Parter, "On Imbedding Graphs in the Sphere", *Journal of Mathematics and Mechanics*, vol. 10, no. 3, pp. 517–523, 1961.

[8] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms", *Journal of Computer and System Sciences*, vol. 13, no. 3, pp. 335–379, Dec. 1, 1976.

[9] Y. Yang. (2016). Graphrel - Explore Your Graph, [Online]. Available: `https://yiboyang.github.io/graphrel/` (visited on 12/12/2017).

[10] G. Team. (2015). Graphonline, [Online]. Available: `http://graphonline.ru/en/` (visited on 12/12/2017).

[11] WolframResearch. (2009). Wolframalpha, [Online]. Available: `https://www.wolframalpha.com` (visited on 12/12/2017).

[12] J. M. Boyer and W. J. Myrvold, "Stop Minding Your p's and q's: A Simplified O (n) Planar Embedding Algorithm.", in *SODA*, vol. 99, 1999, pp. 140–146.

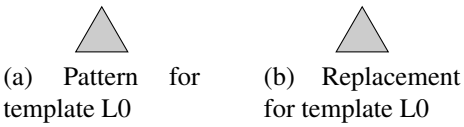
[13] S. Wei-Kuan and H. Wen-Lian, "A new planarity test", *Theoretical Computer Science*, vol. 223, no. 1-2, pp. 179–191, 1999.

[14] S. Even and R. E. Tarjan, "Computing an st-numbering", *Theoretical Computer Science*, vol. 2, no. 3, pp. 339–344, 1975.

Appendix A

Exhaustive list of templates used in PQ-Trees' reduction procedure

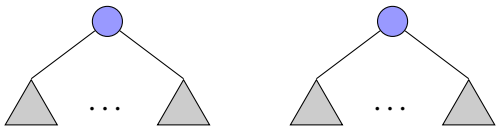
Below is an exhaustive list of figures [1] describing templates described in [8]



(a) Pattern for template L0

(b) Replacement for template L0

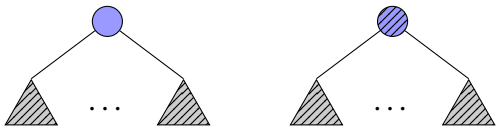
Figure A.1: Template L0 (Leaf)



(a) Pattern for template P0

(b) Replacement for template P0

Figure A.2: Template P0

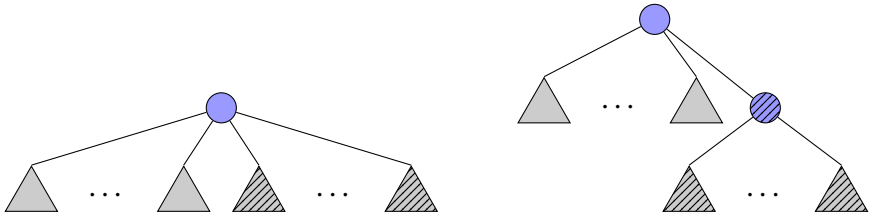


(a) Pattern for template P1

(b) Replacement for template P1

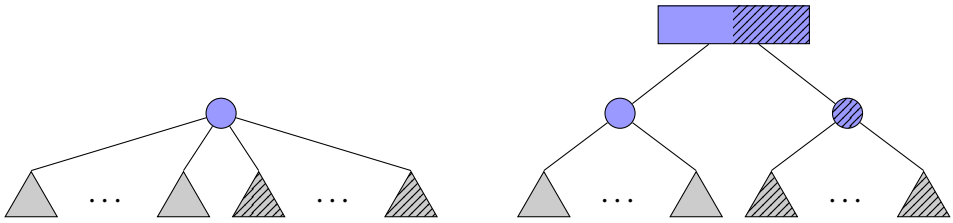
Figure A.3: Template P1

[1] All figures in this chapter use LaTeX code from answer to https://tex.stackexchange.com/questions/125862/drawing-pq-trees by user Qrrbrbirlbel, modified by Hannes Kr. Hannesson



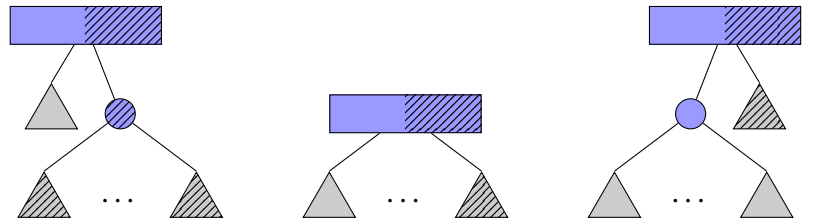
(a) Pattern for template P2 (b) Replacement for template P2

Figure A.4: Template P2



(a) Pattern for template P3 (where P-Node is not $ROOT(T, S)$)

(b) Replacement for template P3

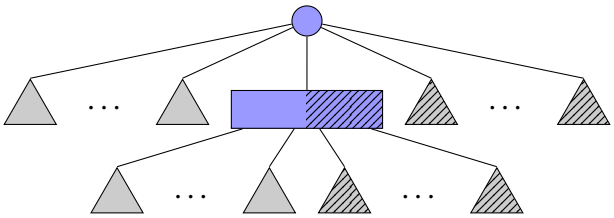


(c) Alternative replacement for template P3

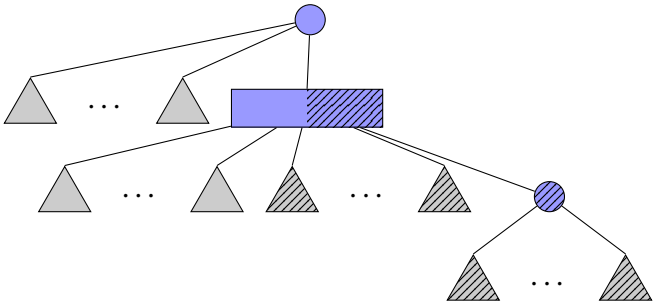
(d) Alternative replacement for template P3

(e) Alternative replacement for template P3

Figure A.5: Template P3

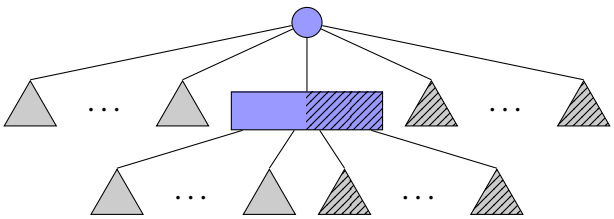


(a) Pattern for template P4 (when P-Node is $ROOT(T, S)$)

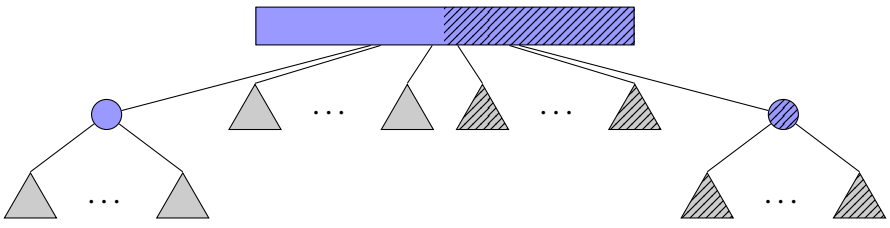


(b) Replacement for template P4

Figure A.6: Template P4

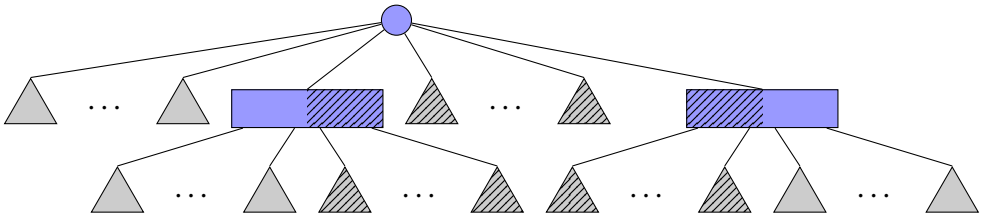


(a) Pattern for template P5 (when P-Node is not $ROOT(T, S)$)

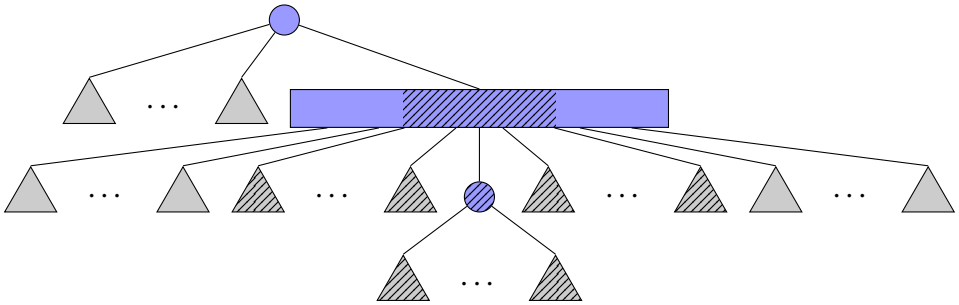


(b) Replacement for template P5

Figure A.7: Template P5

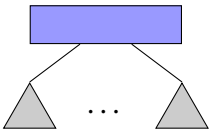


(a) Pattern for template P6 (when P-Node is $ROOT(T, S)$)

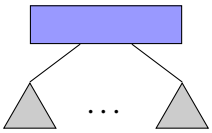


(b) Replacement for template P6

Figure A.8: Template P6

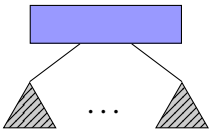


(a) Pattern for template Q0

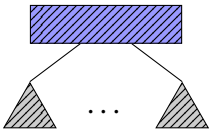


(b) Replacement for template Q0

Figure A.9: Template Q0

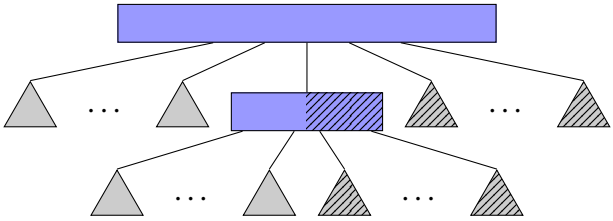


(a) Pattern for template Q1

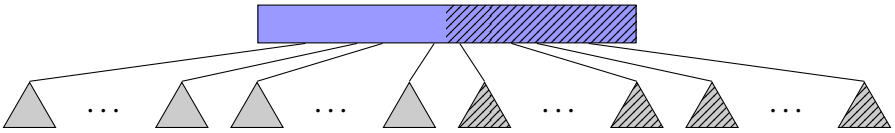


(b) Replacement for template Q1

Figure A.10: Template Q1

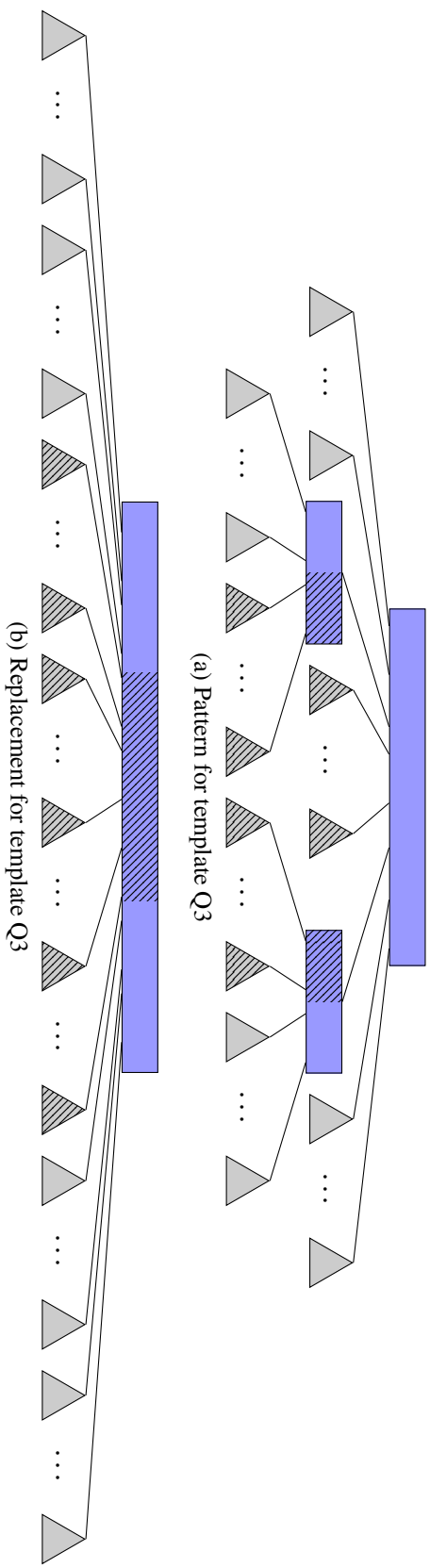


(a) Pattern for template Q2



(b) Replacement for template Q2

Figure A.11: Template Q2



(a) Pattern for template Q3

(b) Replacement for template Q3

Figure A.12: Template Q3

Appendix B

Running times

$log_2(\|E(G)\|)$	log_2(Running time (s))
12	-1.76
13	-0.15
14	2.62
15	5.26
16	7.63
17	9.02

Table B.1: Data collected from running the PQ-Tree method (without st-numbering, after optimization) on cycle graphs, used in Figure 5.1

$\|E(G)\|$	Running time (s)
3	0
6	0
10	0
15	$5 \cdot 10^{-3}$
21	$3 \cdot 10^{-3}$
28	$2 \cdot 10^{-3}$
36	$5 \cdot 10^{-3}$
45	$1 \cdot 10^{-2}$
55	$2.1 \cdot 10^{-2}$
66	$3.5 \cdot 10^{-2}$
78	$8.5 \cdot 10^{-2}$
91	$4.5 \cdot 10^{-2}$
105	$7.3 \cdot 10^{-2}$
120	0.11
136	0.14
153	0.21
171	0.22
190	0.19
210	0.22

231	0.31
253	0.33
276	0.37
300	0.46
325	0.58
351	0.71
378	0.89
406	1.09
435	1.33
465	1.58
496	1.93
528	2.31
561	2.83
595	3.36
630	3.91
666	4.85
703	5.47
741	6.3
780	7.33
820	8.52
861	9.92
903	11.22
946	13.17
990	13.25
1,035	16.22
1,081	18.32
1,128	21.59
1,176	23.44
1,225	27.2
1,275	30.29
1,326	34.37
1,378	39.81
1,431	44.04
1,485	49.51
1,540	55.49
1,596	58.88
1,653	70.26
1,711	74.28
1,770	82.55
1,830	89.25
1,891	103.8
1,953	117.32
2,016	126.23
2,080	139.84
2,145	147.27

Table B.2: Data collected from running the brute-force method on complete graphs

$\lvert E(G)\rvert$	Running time (s)
3	$1.7 \cdot 10^{-2}$
6	$3 \cdot 10^{-3}$
10	$5 \cdot 10^{-3}$
15	$6 \cdot 10^{-3}$
21	$5 \cdot 10^{-3}$
28	$9 \cdot 10^{-3}$
36	$6 \cdot 10^{-3}$
45	$9 \cdot 10^{-3}$
55	$9 \cdot 10^{-3}$
66	$2.2 \cdot 10^{-2}$
78	$1.2 \cdot 10^{-2}$
91	$1.7 \cdot 10^{-2}$
105	$1.7 \cdot 10^{-2}$
120	$5 \cdot 10^{-2}$
136	0.26
153	$2.7 \cdot 10^{-2}$
171	$3.5 \cdot 10^{-2}$
190	$3.6 \cdot 10^{-2}$
210	$2 \cdot 10^{-2}$
231	$2.8 \cdot 10^{-2}$
253	$3 \cdot 10^{-2}$
276	$6.5 \cdot 10^{-2}$
300	$4.4 \cdot 10^{-2}$
325	$5.1 \cdot 10^{-2}$
351	$7.5 \cdot 10^{-2}$
378	$6.5 \cdot 10^{-2}$
406	$7 \cdot 10^{-2}$
435	0.15
465	$9.7 \cdot 10^{-2}$
496	$8.4 \cdot 10^{-2}$
528	0.12
561	$6.1 \cdot 10^{-2}$
595	$6.9 \cdot 10^{-2}$
630	$7.7 \cdot 10^{-2}$
666	0.1
703	0.12
741	0.13
780	0.15
820	0.15
861	0.14
903	0.16
946	0.17
990	0.17
1,035	0.2
1,081	0.24
1,128	0.28

1,176	0.26
1,225	0.29
1,275	0.3
1,326	0.36
1,378	0.35
1,431	0.38
1,485	0.4
1,540	0.42
1,596	0.44
1,653	0.48
1,711	0.49
1,770	0.52
1,830	0.56
1,891	0.59
1,953	0.63
2,016	0.69
2,080	0.72
2,145	0.77
1,953	0.68
2,278	0.94
2,628	1.11
3,003	1.48
3,403	1.87
3,828	2.37
4,278	2.93
4,753	3.61
5,253	4.49
5,778	5.27
6,328	6.75
6,903	7.87
7,503	9.29
8,128	11.49
8,778	14.01
9,453	17.03
10,153	19.55
10,878	23.46
11,628	27.35
12,403	27.56
13,203	34.33
14,028	40.18
14,878	47.69
15,753	54
16,653	59.23
17,578	69.33
18,528	74.4
19,503	85.61
20,503	92.38
21,528	106.86
22,578	117.86

23,653	130.97
24,753	145.45
25,878	148.23
27,028	164.16

Table B.3: Data collected from running the PQ-Tree method on complete graphs (with st-numbering, before optimization)

$\lvert E(G)\rvert$	Running time (s)
0	$1 \cdot 10^{-2}$
1,275	0.16
5,050	0.33
11,325	0.77
20,100	1.72
31,375	3.91
45,150	7.81
61,425	14.57
80,200	25.89
$1.01 \cdot 10^5$	40.14
$1.25 \cdot 10^5$	60.34
$1.52 \cdot 10^5$	90.65
$1.8 \cdot 10^5$	121.79
$2.12 \cdot 10^5$	176.98

Table B.4: Data collected from running the PQ-Tree method on complete graphs (with st-numbering, after optimization)

$\lvert E(G)\rvert$	Running time (s)
5,050	$1.2 \cdot 10^{-2}$
11,325	$2.4 \cdot 10^{-2}$
31,375	$5.9 \cdot 10^{-2}$
45,150	0.1
61,425	$9.3 \cdot 10^{-2}$
80,200	0.14
$1.01 \cdot 10^5$	$9.3 \cdot 10^{-2}$
$1.25 \cdot 10^5$	$7.2 \cdot 10^{-2}$
$1.52 \cdot 10^5$	$6.2 \cdot 10^{-2}$
$1.8 \cdot 10^5$	0.69
$2.12 \cdot 10^5$	$7.2 \cdot 10^{-2}$
$2.45 \cdot 10^5$	$8.8 \cdot 10^{-2}$
$2.82 \cdot 10^5$	$9.8 \cdot 10^{-2}$
$3.2 \cdot 10^5$	$9.1 \cdot 10^{-2}$
$3.62 \cdot 10^5$	0.32

$4.05 \cdot 10^5$	1.19
$4.52 \cdot 10^5$	0.55
$5.01 \cdot 10^5$	0.37
$5.52 \cdot 10^5$	0.17

Table B.5: Data collected from running the PQ-Tree method on complete graphs (without st-numbering, after optimization)

$\lvert E(G)\rvert$	Running time (s)
1	$1 \cdot 10^{-3}$
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	$4 \cdot 10^{-3}$
10	$2.4 \cdot 10^{-2}$
11	$2.9 \cdot 10^{-2}$
12	0.17
13	1.13
14	3.59
15	48.85
16	835.19

Table B.6: Data collected from running the brute-force method on cycle graphs

$\lvert E(G)\rvert$	Running time (s)
1	$8 \cdot 10^{-3}$
101	0.1
201	0.14
301	0.27
401	0.29
501	0.24
601	0.26
701	0.39
801	0.4
901	0.58
1,001	0.66
1,101	0.77
1,201	0.88
1,301	1.03

1,401	1.23
1,501	1.49
1,601	1.79
1,701	1.81
1,801	2.22
1,901	3.35
2,001	3.37
2,101	2.71
2,201	3.18
2,301	3.71
2,401	3.61
2,501	4.24
2,601	4.28
2,701	4.78
2,801	4.98
2,901	5.63
3,001	5.83
3,101	6.42
3,201	6.86
3,301	7.4
3,401	8.64
3,501	8.7
3,601	9.26
3,701	9.54
3,801	10.47
3,901	10.22
4,001	11.76
4,101	11.83
4,201	12.56
4,301	13.43
4,401	13.59
5,001	21.15
6,001	31.12
7,001	46.33
8,001	61.45
9,001	86.86

Table B.7: Data collected from running the PQ-Tree method on cycle graphs (with st-numbering, before optimization)

$\lvert E(G)\rvert$	Running time (s)
1	$7 \cdot 10^{-3}$
201	$2.2 \cdot 10^{-2}$
401	$2.6 \cdot 10^{-2}$
601	$2.4 \cdot 10^{-2}$
801	$1.6 \cdot 10^{-2}$

1,001	$2.4 \cdot 10^{-2}$
1,201	$3.3 \cdot 10^{-2}$
1,401	$3.1 \cdot 10^{-2}$
1,601	$3.6 \cdot 10^{-2}$
1,801	0.12
2,001	0.14
2,201	0.1
2,401	$7.7 \cdot 10^{-2}$
2,601	0.1
2,801	$9.7 \cdot 10^{-2}$
3,001	$8.5 \cdot 10^{-2}$
3,201	0.16
3,401	0.15
3,601	0.15
3,801	0.2
4,001	0.14
4,201	0.19
4,401	0.28
4,601	0.22
4,801	0.15
5,001	0.18
5,201	0.19
5,401	0.24
5,601	0.27
5,801	0.25
6,001	0.27
6,201	0.37
6,401	0.29
6,601	0.35
6,801	0.37
7,001	0.33
7,201	0.35
7,401	0.37
7,601	0.42
7,801	0.38
8,001	0.41
8,201	0.42
8,401	0.51
8,601	0.54
8,801	0.5
9,001	0.55

Table B.8: Data collected from running the PQ-Tree method on cycle graphs (with st-numbering, after optimization)

$\lvert E(G)\rvert$	Running time (s)

4,096	0.31
8,192	0.92
16,384	6.1
32,768	37.84
65,536	196.7
$1.31 \cdot 10^5$	518.75

Table B.9: Data collected from running the PQ-Tree method on cycle graphs (without st-numbering, after optimization)

$\lvert E(G)\rvert$	Running time (s)
3	$2 \cdot 10^{-3}$
6	0
9	$9 \cdot 10^{-3}$
12	0.21
15	5.84
18	1,713.33

Table B.10: Data collected from running the brute-force method on gear graphs

$\lvert E(G)\rvert$	Running time (s)
3	$2 \cdot 10^{-3}$
303	0.62
603	0.9
903	2.04
1,203	4.86
1,503	9.52
1,803	16.48
2,103	26.22
2,403	38.65
2,703	54.62
3,003	74.54
3,303	103.15
3,603	129.38
3,903	166.06
4,203	223.71
4,503	295.16
4,803	370.88
5,103	448.76
5,403	535.31
5,703	636.17
6,003	756.83
6,303	884.16

6,603	1,023.99
6,903	1,179.85
7,203	1,345.85
7,503	1,530.74
7,803	1,727.23

Table B.11: Data collected from running the PQ-Tree method on gear graphs (with st-numbering, before optimization)

$\|E(G)\|$	Running time (s)
3	$1.4 \cdot 10^{-2}$
903	0.21
1,803	0.23
2,703	0.53
3,603	0.7
4,503	0.73
5,403	0.8
6,303	1.13
7,203	1.9
8,103	2
9,003	2.8
9,903	3.31
10,803	3.87
11,703	4.75
12,603	5.63
13,503	7.75

Table B.12: Data collected from running the PQ-Tree method on gear graphs (with st-numbering, after optimization)

Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is