TU WIEN Informatics

Der Left-Right Planaritätstest

Vergleich von Varianten und Implementierungen

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Sebastian Wodniansky-Wildenfeld

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Martin Nöllenburg
Mitwirkung: Dr. Simon Dominik Fink

Wien, 11. Februar 2025

Sebastian Wodniansky-Wildenfeld

Martin Nöllenburg

TU WIEN Informatics

The Left-Right Planarity Test

Comparing Variants and Implementations

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Business Informatics

by

Sebastian Wodniansky-Wildenfeld

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Martin Nöllenburg
Assistance: Dr. Simon Dominik Fink

Vienna, February 11, 2025

Sebastian Wodniansky-Wildenfeld

Martin Nöllenburg

Erklärung zur Verfassung der Arbeit

Sebastian Wodniansky-Wildenfeld

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 11. Februar 2025

Sebastian
Wodniansky-Wildenfeld

Kurzfassung

Die Planaritätstestung ist das Problem, zu bestimmen, ob ein gegebener Graph in der Ebene so gezeichnet werden kann, dass sich keine Kanten überschneiden. Sie ist eine fundamentale Fragestellung in der Graphentheorie mit Anwendungen in der Schaltkreisentwicklung, geografischen Informationssystemen und der Netzvisualisierung. Für diese Aufgabe wurden verschiedene effiziente Algorithmen entwickelt, die unterschiedliche strukturelle und kombinatorische Techniken zur Verifikation der Planarität verwenden. Eine dieser Methoden ist der Left-Right-Ansatz zur Planaritätstestung, der erstmals von Rosenstiehl und de Fraysseix (1982) vorgestellt und später von de Fraysseix, Ossona de Mendez und Rosenstiehl (2006) weiterentwickelt wurde. Brandes (2008) schlug zusätzliche Vereinfachungen vor, deren praktische Effizienz jedoch bisher nicht untersucht wurde. In dieser Arbeit wird die Version von Brandes implementiert und systematisch mit früheren Implementierungen sowie mit aktuellen Planaritätstest-Algorithmen verglichen. Die Analyse unterscheidet zwischen Verfeinerungen, die die Effizienz verbessern, und solchen, die vor allem die Lesbarkeit erleichtern. Durch eine Kombination aus theoretischer Untersuchung und empirischer Evaluierung wird der Einfluss dieser Modifikationen geklärt und die Anwendbarkeit des Left-Right-Ansatzes für praktische Anwendungen bewertet.

Abstract

Planarity testing is the problem of determining whether a given graph can be drawn in the plane without any edge crossings. It is a fundamental question in graph theory with applications in circuit design, geographic information systems, and network visualization. Several efficient algorithms have been developed for this task, each employing different structural and combinatorial techniques to verify planarity. One such approach is the Left-Right planarity testing method, first introduced by Rosenstiehl and de Fraysseix (1982) and later refined by de Fraysseix, Ossona de Mendez, and Rosenstiehl (2006). Brandes (2008) proposed further simplifications, but their practical efficiency has not been evaluated. This work implements Brandes' version and systematically compares it to previous implementations and state-of-the-art planarity testing algorithms. The analysis distinguishes between refinements that improve efficiency and those that primarily enhance readability. Through both theoretical and empirical evaluations, this study clarifies the impact of these modifications and assesses the suitability of the Left-Right approach for practical applications.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Related Work 1
1.2 Motivation 2
1.3 Thesis Structure 3

2 Preliminaries **5**
2.1 Graph fundamentals 5
2.2 Depth-First Search and Terminology 6
2.3 Characterization of Planar graphs 8
2.4 Definitions and Notation for Planarity Testing 10

3 Left-Right Approach **15**
3.1 Overview 15
3.2 Orientation Phase 15
3.3 Testing Phase 16
3.4 Embedding Phase 22

4 Literature Review **25**
4.1 Glossary 26
4.2 Comparison of similar terms and concepts 27
4.3 Algorithm Comparison 30

5 Implementation Details **35**
5.1 Our Brandes Implementation 35
5.2 The PIGALE Implementation 37

6 Empirical Evaluation **47**
6.1 Test data 47

6.2 Testing Set-up 48
6.3 Results 49

7 Discussion **59**
7.1 Summary and Interpretation of Key Findings 59
7.2 Future Research Directions 60

Overview of Generative AI Tools Used **61**
Notebook LM 61
ChatGPT 61

List of Figures **63**

Bibliography **65**

CHAPTER 1

Introduction

Graph planarity testing is a fundamental problem in graph theory and algorithm design. A graph is considered planar if it can be drawn in the plane without any edge crossings. Planarity testing has broad applications in areas such as circuit design, geographic information systems, and network visualization. Despite its importance, many of the existing planarity testing algorithms are complex and require intricate data structures, making their implementation and understanding challenging.

Two primary strands of planarity testing algorithms exist: vertex-addition methods, pioneered by Lempel, Even, and Cederbaum (1967) [Lem67], and path-addition methods, refined by Hopcroft and Tarjan [HT74]. While these methods are well-documented, they remain difficult to comprehend and implement. The Left-Right Planarity Testing approach extends the path-addition method by introducing a structured characterization of planarity based on cycle orientations in a depth-first search (DFS) tree. First introduced by de Fraysseix and Rosenstiehl (1982, 1985) [dFR82, dFR85] and further refined by de Fraysseix, Ossona de Mendez, and Rosenstiehl (2006, 2008, 2012) [dFdMR06, dF08, dFdM12], the Left-Right approach characterizes planarity based on cycle orientations in a depth-first search (DFS) tree. Unlike other methods, it does not require complex embedding phases or biconnected component treatments, making it both elegant and efficient.

1.1 Related Work

One of the pivotal figures in planarity testing is Kazimierz Kuratowski, who provided a fundamental characterization of planar graphs in 1930 [Kur30]. Kuratowskis theorem states that a graph is planar if and only if it does not contain a subgraph homeomorphic to the complete graph on five vertices (K_5) or the complete bipartite graph with three vertices in each partition ($K_{3,3}$). This theorem laid the theoretical foundation for planarity testing and influenced the development of algorithmic solutions in the decades that followed.

Planarity testing has been extensively studied, with several polynomial and linear-time algorithms developed over the decades. The first polynomial-time algorithm for planarity testing was introduced by Auslander and Parter in 1961 [AP61], marking a significant milestone in the field. Hopcroft and Tarjan [HT74] later introduced the first linear-time algorithm, building on the work of Auslander and Parter with an iterative, DFS-based approach to improve efficiency. Their method builds a DFS tree and incrementally determines whether a planar embedding is possible. Around the same time, Lempel, Even, and Cederbaum (1967) [Lem67] introduced the vertex-addition approach, which incrementally constructs a planar embedding by adding one vertex at a time. These two approaches have been the foundation of most modern planarity testing algorithms.

Subsequent improvements focused on practical efficiency and ease of implementation. Booth and Lueker (1976) [BL76] introduced PQ-trees as a data structure to simplify the embedding process, while Mehlhorn and Mutzel (1996) [MM96] provided further refinements. The Boyer-Myrvold algorithm (2004) [BM99, BM06] introduced a new method for embedding graphs in linear time, which has since become one of the most commonly used planarity tests in practical applications.

In contrast to these established methods, the Left-Right approach, first described by de Fraysseix and Rosenstiehl [dFR82], remains relatively unexplored. De Fraysseix, Ossona de Mendez, and Rosenstiehl (2006, 2008, 2012) [dFdMR06, dF08, dFdM12] later improved their own algorithm, refining its structure and efficiency. Unlike traditional approaches, it relies on an ordering of back edges in a DFS tree to determine whether a consistent planar embedding exists. Recent work by Brandes (2008) [Bra09] provided a comprehensive exposition of this method, demonstrating its simplicity and efficiency. Despite its potential, the Left-Right approach has not been widely implemented or studied in comparison to more traditional methods. However, it has an efficient implementation in PIGALE (2002) [dFdM02], demonstrating its practical feasibility.

1.2 Motivation

The motivation for this thesis stems from the need to evaluate and refine an alternative approach to planarity testing. The Left-Right approach, originally developed by Rosenstiehl and further refined by Brandes, offers a simplification of existing algorithms. However, its real-world applicability and practical efficiency remain largely unexplored.

This work aims to implement the pseudo-code provided by Brandes, assess its performance compared to earlier implementations, and determine how it optimizes the original method. By systematically comparing its theoretical improvements with actual implementation results, we seek to provide insights into whether these refinements translate into measurable efficiency gains.

Furthermore, this thesis evaluates the Left-Right approach in the broader landscape of planarity testing algorithms. By comparing it to state-of-the-art methods, we analyze whether it holds practical value beyond theoretical considerations. If proven effective, this

approach could offer a more accessible and efficient alternative to existing solutions. If not, our analysis will clarify its limitations and contribute to the refinement of planarity testing methodologies.

A key objective of this research is to clarify the evolution of the Left-Right approach and determine whether Brandes refinements offer substantial algorithmic improvements or merely serve to enhance readability and comprehensibility. By systematically dissecting the modifications introduced by Brandes, we aim to assess whether they lead to tangible efficiency gains or primarily provide a more accessible exposition of the algorithm.

1.3 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 introduces essential graph theory concepts, including fundamental definitions, depth-first search terminology, and planarity characterizations. Chapter 3 presents the Left-Right approach, outlining its algorithmic structure and theoretical foundation. Chapter 4 compares different versions of the Left-Right algorithm in literature, analyzing how modifications impact performance and clarity. Chapter 5 details our implementation of Brandes pseudo-code and compares it to previous implementations, particularly those found in PIGALE. Chapter 6 evaluates the efficiency of the implemented algorithm, presenting test data, methodology, and results. Finally, Chapter 7 summarizes the key findings of this thesis and provides suggestions for future research directions.

CHAPTER 2

Preliminaries

This chapter provides the fundamental concepts necessary for understanding planarity testing. We introduce basic graph theory definitions, depth-first search (DFS) as a key traversal method, and key planarity characterizations such as Kuratowskis theorem and Eulers formula. Additionally, we establish essential notation and definitions used throughout the thesis. These preliminaries form the theoretical foundation for the algorithms analyzed in later chapters.

2.1 Graph fundamentals

A *graph* $G = (V, E)$ is defined as an ordered pair consisting of a set of *vertices* V and a set of *edges* E, where $E \subseteq \{\{u, v\} \mid u, v \in V\}$. The nature of these connections depends on whether the graph is *directed* or *undirected*.

In an *undirected graph*, each edge is a *set* $\{u, v\}$ with $u, v \in V$, meaning that there is no inherent direction between the two connected vertices. This implies that $\{u, v\} = \{v, u\}$, making the relationship symmetric.

In contrast, a *directed graph* assigns directionality to edges. Here, each edge is an *ordered pair* (u, v), where u is the *source vertex* and v is the *target vertex*. Unlike in undirected graphs, direction matters, so $(u, v) \neq (v, u)$ in general.

A *simple graph* is a graph that does not allow *multiple edges* between the same pair of vertices and does not contain *loops*. Formally, for all $u, v \in V$, a simple graph satisfies $u \neq v$ and the edge set contains at most one pair $\{u, v\}$.

In the algorithm discussed later, the input graph is always assumed to be a simple undirected graph, as edge direction, loops, and multiple edges do not affect planarity.

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$. This means that G' consists of a subset of the vertices and edges of G, preserving the relationships between vertices as they exist in G.

A graph $G = (V, E)$ is said to be *connected* if, for every pair of vertices $u, v \in V$, there exists a path P between them, where P is a sequence of edges $\{u_1, u_2\}, \{u_2, u_3\}, \ldots, \{u_{k-1}, u_k\}$ with $u_1 = u$ and $u_k = v$. If no such path exists for some pair of vertices, the graph consists of multiple *connected components*, each of which is a maximal connected subgraph of G.

A key structural property of graphs is *biconnectivity*. A graph G is *biconnected* if it remains connected after the removal of any single vertex. More formally, a graph is biconnected if for every vertex $v \in V$, the graph $G' = (V - \{v\}, E')$, remains connected. A *cut vertex* (or articulation point) is a vertex $v \in V$ whose removal results in the graph becoming disconnected, i.e., the number of connected components increases in the graph $G - v$.

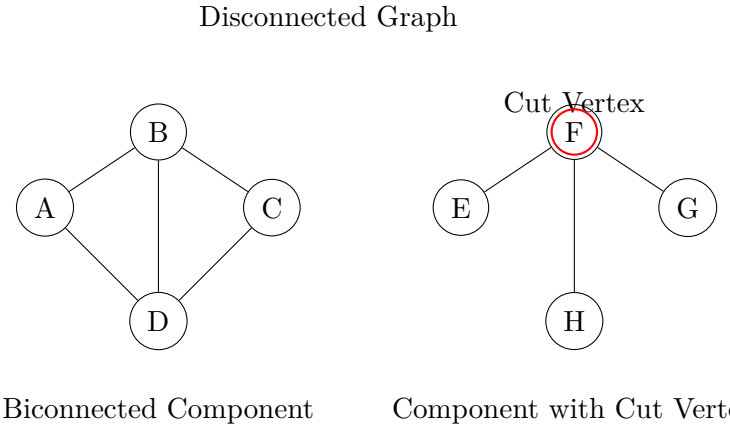


Figure 2.1: Illustration of Connected Components, Biconnectivity, and a Cut Vertex.

2.2 Depth-First Search and Terminology

Depth-First Search (DFS) is a foundational graph traversal algorithm that plays a crucial role in many graph-related problems, including planarity testing. This chapter defines DFS, introduces the concept of a DFS tree, and explains its significance in the context of graph algorithms.

Definition *Depth-First Search (DFS)* is an algorithm designed to explore the vertices and edges of a graph in a systematic manner. Starting from a specified root vertex, the algorithm follows a depth-first strategy, exploring as far along each branch as possible before backtracking.

Formally, let $G = (V, E)$ represent a graph with vertices V and edges E. The DFS traversal can be described as follows:

- Begin at a designated vertex $v \in V$.
- Mark v as visited.
- Recursively visit each unvisited neighbor by performing the above steps

If G is disconnected, DFS can be applied to each connected component separately. The traversal order of vertices and edges generated by DFS forms the basis for constructing a DFS tree.

In the context of DFS, the traversal alternates between two distinct phases:

Discovery During this phase, the algorithm visits a vertex for the first time, marks it as visited, and explores all its unvisited neighbors. Edges leading to these unvisited neighbors are classified as *tree edges*, forming part of the DFS tree. Additionally, if the algorithm encounters a neighbor that is already visited is is therefore an ancestor in the DFS tree, the edge is classified as a *back edge*.

Backtracking Once all neighbors of a vertex have been processed during the discovery phase, the algorithm backtracks to its predecessor in the DFS tree. This ensures that all possible paths emanating from previously visited vertices are fully explored.

By alternating between discovery and backtracking, DFS ensures a systematic exploration of all vertices and edges in the graph, organizing them into a hierarchical structure.

DFS Tree A *DFS tree* is a spanning tree of a connected component of the graph G constructed during the DFS traversal. It organizes the vertices and edges of the graph into a hierarchical structure, reflecting the order in which they are discovered.

The DFS tree contains:

- *Tree Edges:* Edges added to the DFS tree during the traversal, connecting a vertex to its unvisited neighbor. We refer to the set of all tree edges as T.
- *Back Edges:* Edges that point back to an ancestor in the DFS tree. These edges are not part of the DFS tree but are crucial for detecting cycles and understanding the graph's structure. We denote the set of all back edges as B.

The direction of these edges is assigned according to the traversal order: tree edges are directed upwards in the DFS tree, while back edges are directed downwards, as illustrated in Figure 2.3.

This assignment of directions effectively transforms an undirected graph with edge set E into a directed graph with an oriented edge set $\vec{E}$. In this oriented graph, the set of all outgoing edges of a vertex $v \in V$ is denoted as:

$$E^+(v) = \{(v, w) \in \vec{E} \mid w \in V\}$$

such that the entire edge set is expressed as:

$$\vec{E} = \bigcup_{v \in V} E^+(v).$$

For any connected component, the DFS tree has the following properties:

- It includes all vertices of the component.
- It contains exactly $|V| - 1$ tree edges for $|V|$ vertices.
- Each vertex, except the root, has exactly one parent in the DFS tree.

If G consists of multiple connected components, the collection of all DFS trees forms a *DFS forest*. In the context of DFS, the relationships between vertices and edges in a graph can be captured through the definitions of paths and cycles. A *tree path* is the unique sequence of edges connecting two vertices u and v in the DFS tree. If the path may be empty, this is denoted $u \to^* v$, while $u \to^+ v$ represents a path that contains at least one edge. Using these tree paths, we can define *fundamental cycles*. A *back edge*, denoted $v \curvearrowright w$, connects a vertex v to one of its ancestors w in the DFS tree. A fundamental cycle $C(v \curvearrowright w)$ is formed by combining a back edge with the corresponding tree path, formally given by:

$$C(v \curvearrowright w) = w \to^+ v \cup \{v \curvearrowright w\}.$$

Two cycles are said to *overlap* if they share a common tree path in the DFS tree. Formally, let C_1 and C_2 be two cycles. If there exists a non-empty tree path $u \to^+ v$ that is a subpath of both C_1 and C_2, then C_1 and C_2 are overlapping. For example, in Figure 2.3, the cycles $C(v_4 \curvearrowright v_2)$ and $C(v_5 \curvearrowright v_2)$ overlap along the shared tree path $v_2 \to^+ v_3$.

In a DFS tree, the *height* of a vertex is defined as its distance from the root. The root is considered the lowest vertex in the tree, and its height is set to zero. The height of any other vertex v is the number of edges in the path from the root to v. For example, in Figure 2.3, the height of the root v_1 is 0. The vertices v_4 and v_5 both have a height of 3, as their paths from the root each consist of three edges.

2.3 Characterization of Planar graphs

A *planar graph* is a graph that can be drawn in the plane such that no edges intersect except at their endpoints. Formally, a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, is said to be planar if there exists a drawing of G in the Euclidean plane where each edge is represented by a continuous curve, and no two edges share any points other than their endpoints (i.e., the vertices they connect). Such a drawing is called a *plane embedding* of the graph.

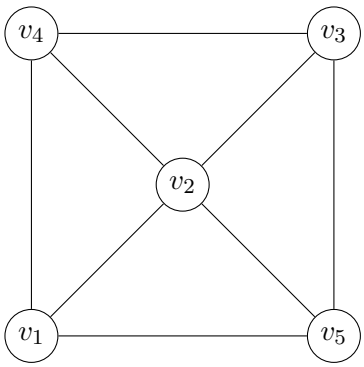


Figure 2.2: Simple undirected planar graph

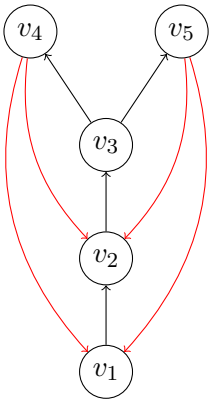


Figure 2.3: DFS-tree of the graph, with tree edges in black and back edges in red

A key characterization of planar graphs is provided by *Kuratowski's Theorem* [Kur30], which states that a graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 (the complete graph on five vertices) or $K_{3,3}$ (the complete bipartite graph on two sets of three vertices). These two graphs are the simplest examples of non-planar graphs, meaning that it is impossible to draw them in the plane without at least one edge crossing. Kuratowskis Theorem serves as a fundamental tool for determining whether a given graph is planar or non-planar.

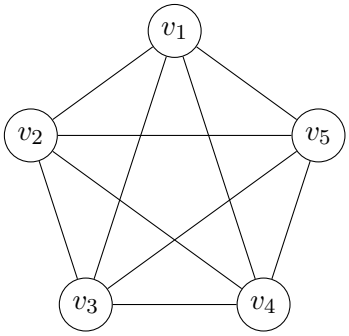


Figure 2.4: Complete graph K_5

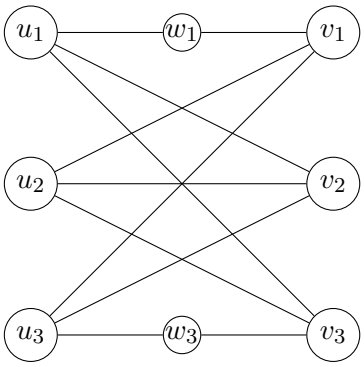


Figure 2.5: Subdivision of $K_{3,3}$

Planar graphs also satisfy *Eulers Formula* [Eul22], which provides a relationship between the number of vertices n, edges m, and faces f in any connected planar graph. The faces of a planar graph are the regions into which the plane is divided by its embedding, where each bounded region is enclosed by edges, and an additional unbounded outer region exists:

$$n - m + f = 2.$$

Any graph that does not satisfy this formula cannot be planar. From Euler's Formula, we can derive a bound on the number of edges in a planar graph. Since each edge can belong to at most two faces and each face is bounded by at least three edges, it follows that $f \leq \frac{2m}{3}$. Substituting this into Eulers Formula leads to the result that for planar graphs with $n > 2$ vertices, the number of edges is limited by:

$$m \leq 3n - 6.$$

This result indicates that planar graphs cannot be arbitrarily dense, ensuring that their edge count is bounded relative to the number of vertices.

2.4 Definitions and Notation for Planarity Testing

This section introduces key definitions and notation that will be used in the context of the left-right planarity testing approach. The definitions will be applied to a depth-first search (DFS) tree to analyze whether a graph can be embedded in the plane without edge crossings.

Definition 1. A *back edge* $x \curvearrowright u \in B$ is called a *return edge* for a tree edge $v \to w \in T$ if its source node x is either a descendant of w or w itself, and its target node u is an ancestor of v. Formally, for a tree edge $v \to w \in T$, a back edge $x \curvearrowright u \in B$ is a return edge if there exists a path $u \xrightarrow{+} v \to w \xrightarrow{*} x \curvearrowright u$, where x is a descendant of w.

Definition 2. We define the *return points* of a tree edge as the target nodes of its return edges. For a back edge $u \curvearrowright v \in B$, we define its target node v as its single return point.

An example illustrating return edges and return points is shown in Figure 2.6.

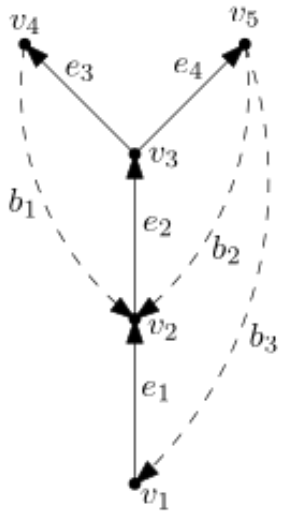


Figure 2.6: Example of return edges. Note that b_2 and b_3 are return edges for e_4, but only b_3 is a return edge for e_2.

Definition 3. The *lowpoint* of an edge is defined as the return point with the smallest height in the DFS tree, if any exist; otherwise, it is the source of the edge. Consequently, the lowpoint of a back edge is always its target node, as it is the only return point associated with the edge.

Formally, for an edge $e \in E$, the lowpoint $\text{low}(e)$ is defined as:

$$\text{lowpoint}(e) = \begin{cases} \min\{v \in V : \exists (u, v) \text{ is a return edge of } e, & \text{if } e \text{ is a tree edge,} \\ y, & \text{if } e = (x, y) \text{ is a back edge.} \end{cases}$$

In the later explained algorithm, we use the lowpoint and its integer height interchangeably.

Therefore in Figure 2.6 the lowpoints for e_2 and e_4 are the height of v_1, while the lowpoint for e_3 is the height of v_2.

For each edge e, the its return edge that returns to the lowpoint of e is called its *lowpoint edge*. This is the back edge that leads to the vertex with the smallest height reachable from e. In particular, for a back edge, the lowpoint edge is always the back edge itself.

Definition 4. We call a tree edge *chordal* if it has more than one return point.

Therefore in Figure 2.6 e_4 is chordal, but e_3 is not.

Definition 5. A *fork* consists of a *branching vertex* v also referred to as a *branching point*, its incoming tree edge e, and two outgoing edges e_1 and e_2. For every two overlapping fundamental cycles, there exists exactly one fork that induces these cycles.

To determine whether a graph is planar, it is crucial to establish, for each fork, which of its outgoing edges is assigned to the left and which to the right. This assignment ensures a consistent arrangement of edges around each vertex, ultimately defining a proper embedding of the graph.

In Figure 2.6:

- The branching vertex v_3 serves as the branching point.
- The incoming tree edge is $e_2 = (v_2, v_3)$.
- The outgoing edges are $e_3 = (v_3, v_4)$ and $e_4 = (v_3, v_5)$.

Definition 6 (Brandes [Bra09])**.** Let $G = (V, T \cup B)$ be a DFS-oriented graph. A partition $B = L \cup R$ of its back edges into two classes, referred to as left and right, is called a *left-right partition*, or LR partition, if for every fork consisting of $u \to v \in T$ and $e_1, e_2 \in E^+(v)$:

1. All return edges of e_1 ending strictly higher than $\text{lowpt}(e_2)$ belong to one class, and

2. All return edges of e_2 ending strictly higher than lowpt(e_1) belong to the other class.

Definition 7. A *constraint* represents a relationship between two back edges that determines if they belong to the same or to different classes.

We distinguish between two types of constraints: *same-constraint* and *different-constraint*, which are defined as follows:

- A different-constraint specifies that two edges must belong to different classes (e.g., left and right) in the partition.
- A same-constraint specifies that two edges must belong to the same class (e.g., both left or both right) in the partition.

Constraints arise during the processing of forks and ensure that the left-right partition of back edges maintains a consistent and valid embedding.

Consider a fork with branching point v, its incoming edge e, and two outgoing edges e_1 and e_2. Let b_1 be a return edge for e_1 and b_2 be a return edge for e_2. The two edges b_1 and b_2 are subject to:

- A *different-constraint*, if lowpt(e_1) < lowpt(b_2) and lowpt(e_2) < lowpt(b_1).
- A *same-constraint*, if an edge $e' = (v', w)$ exists for which $v > v' > \max\{\text{lowpt}(b_1), \text{lowpt}(b_2)\}$ and $\text{lowpt}(e') < \min\{\text{lowpt}(b_1), \text{lowpt}(b_2)\}$.

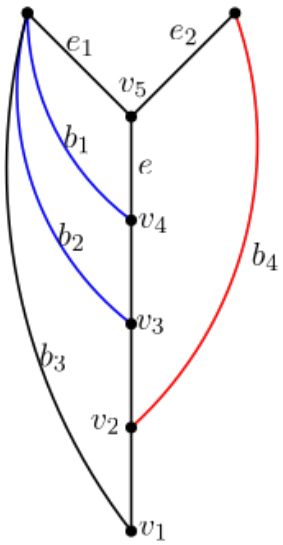


Figure 2.7: Example graph showing same-constraints in the same color and different-constraints in different colors.

Giving an example based on Figure 2.7, we can see a fork with branching point v_5, its incoming edge e, and two outgoing edges e_1 and e_2. In this case, there are four back edges: b_1, b_2, and b_3, which are induced by e_1, and b_4, which is induced by e_2.

From the image, we can see that the back edges b_1 and b_2 have their lowpoints higher than the lowpoint of b_4, which is the lowpoint edge of e_2. Therefore, b_1 and b_2 are subject to a *same-constraint*.

Additionally, back edges b_2 and b_4 are subject to a *different-constraint*, as:

$$\text{lowpt}(e_1) = v_1 < \text{lowpt}(b_4) = v_2, \quad \text{and} \quad \text{lowpt}(e_2) = v_2 < \text{lowpt}(b_2) = v_3.$$

The same applies to b_1 and b_4.

If any two back edges are subject to both a same-constraint and a different-constraint at the same time, the graph is not planar.

Theorem 1 (Left-Right Planarity Criterion [Bra09], see also [dFR85])**.** A graph is planar if and only if it admits an LR partition.

Definition 8. An *LR Partition* is called an *aligned LR Partition* if every return edge of a tree edge e that returns to the lowpoint of e is assigned to the same side.

This alignment is necessary because if even one of these return edges is part of an LR constraint, then all others returning to the same lowpoint must also be involved in that constraint. To simplify the process and avoid inconsistencies, all such return edges are always assigned to the same sideeven if none of them are directly constrained. [Bra09]

While the alignment of the partition is not necessary to test planarity it can be used to improve efficiency by only testing for an aligned LR Partition.

Intuition Left-Right Approach The main idea of the LR algorithm is to place each back edge either on the left or on the right of the DFS tree. Consider two back edges $u \curvearrowright v$ and $u' \curvearrowright v'$, where $v' < v < u' < u$. If both back edges were drawn on the same side of the shared path, they would cross. Therefore, they have to be drawn on different sides to maintain planarity.

To avoid crossing cycles, cycles must either be placed on different sides or nested, where one cycle is placed inside the other. Nesting is possible if the edges form the following structure: $v \leq v' < u' \leq u$. If this condition cannot be satisfied for each overlapping cycle, the graph is not planar. Finally, to compute the planar embedding, all incident edges for each node are ordered, resulting in a drawing where overlapping cycles are either nested or placed on opposite sides.

CHAPTER 3

Left-Right Approach

The Left-Right approach described in this section is based on the exposition by Brandes [Bra09], incorporating his provided pseudo-code. This approach is a more recent version of the original Left-Right method introduced by Rosenstiehl and de Fraysseix in 1982 [dFR82], which they simplified in 2006 [dFdMR06].

3.1 Overview

The Left-Right approach is divided into three phases:

- **Orientation Phase**: In this phase, some information for edges and vertices is collected.
- **Testing Phase**: This phase contains most of the actual logic for planarity testing, determining if the graph is planar by computing an LR partition.
- **Embedding Phase**: In this phase, the planar embedding of the examined graph is computed.

All three phases individually perform a depth-first-search of the graph, however the order of the traversal may differ for each phase, because edges around each node will be reordered according to the information computed in the previous phase.

3.2 Orientation Phase

The orientation phase begins with a depth-first search (DFS) traversal for each connected component of the graph. During traversal, the algorithm maintains a list of the roots of each DFS spanning tree, referred to as *Roots*, and stores the distance of each vertex from

its root in a *height* array for further processing. This step assigns a direction to all edges and computes key attributes needed for planarity testing.

Each edge is assigned a *lowpoint* and *nesting depth*, which play a crucial role in the structural analysis of the graph. During the discovery phase of the DFS, the target node of each edge is stored as its initial lowpoint and lowpoint2, with back edges directly taking their target node as their actual lowpoint. As the algorithm backtracks, a parent edge updates its lowpoint by inheriting the smallest lowpoint value from its successor edges, but only if this value is smaller than its current lowpoint. The same update rule applies to lowpoint2. If, after backtracking, a tree edge has a lowpoint2 that differs from its initial value, it is classified as chordal. This process ensures that the lowpoint, as defined in Definition 2, correctly represents the lowest return point of an edge. Eventually, the nesting depth can be derived from the lowpoint, distinguishing chordal from non-chordal edges with the following formula:

$$\text{nesting\_depth}(e) = \begin{cases} \text{lowpoint}(e), & \text{if } e \text{ is non-chordal,} \\ \text{lowpoint}(e) + 1, & \text{if } e \text{ is chordal.} \end{cases}$$

We say that $e_1 \prec e_2$ if nesting_depth$[e_1] >$ nesting_depth$[e_2]$. If we consider this order for all outgoing edges of a vertex v in $E^+(v)$, we refer to it as the *partial order* around vertex v. This relationship ensures that the nesting order reflects the traversal order of the DFS tree in the testing phase.

This phase operates independently on each connected component without requiring the graph to be connected.

3.3 Testing Phase

The testing phase aims to resolve conflicts from overlapping cycles and determine whether they should be oriented on the same or opposite sides. Overlapping cycles share a common tree path, with their highest node called the branching point. Resolving these conflicts at each fork determines if an aligned LR partition exists, proving the graph's planarity.

This phase involves a second traversal of the DFS tree, processing outgoing edges in the order of their nesting depth. If conflicts cannot be resolved, the traversal halts, indicating non-planarity.

3.3.1 Data Structures in Brandes' Algorithm

Brandes' algorithm relies on several key data structures to efficiently manage edges, enforce constraints, and maintain the left-right partitioning required for planarity testing. These data structures include:

1. *ref*: The *ref* array stores the reference edge for each edge, defining their relative relationships and helping determine their sides. Since the side of most edges is not defined absolutely but depends on another edge, this reference is crucial for maintaining the correct structure.

2. *side*: The *side* array indicates the side to which each edge belongs:

   - A value of -1 indicates the edge is on the opposite side of its reference edge.
   - A value of $+1$ indicates the edge is on the same side as its reference edge.
   - If an edge does not have a reference ($ref[e] = \varnothing$), the value of *side* directly specifies whether the edge belongs to the left side (-1) or the right side ($+1$).

3. *stack*: The stack is a critical data structure in Brandes' algorithm, designed to track and manage conflicts between back edges during the traversal.

   - **Intervals**: An interval is defined by its *highest* and *lowest* edges, denoted as $[e_{\text{low}}, e_{\text{high}}]$. Intermediate edges within an interval are not stored directly; instead, references to these edges are maintained in the *ref* array. This compact representation reduces memory usage while preserving the necessary information for resolving constraints and for drawing the planar graph in a later step.
   - **Stack Units**: A stack unit is a pair of intervals, represented as a left interval and a right interval. These intervals represent conflicting edges, with edges subject to a *different-constraint* placed in separate intervals of the same stack unit. If no edges exist for one side, the respective interval is marked as *null*.

4. *stack_bottom*: In the array *stack_bottom*, we record the top stack unit at the moment an edge e is traversed during the discovery phase. This allows the algorithm to efficiently process only the relevant successor edges during backtracking, as all stack units above $stack\_bottom[e]$ correspond to edges originating from the subtree rooted at e.

5. *lowpoint_edge*: The *lowpoint_edge* array stores the back edge leading to the lowpoint of each tree edge. A back edge is always its own lowpoint edge. Storing lowpoint edges separately is necessary for checking constraints, as they directly refer to these edges, and for determining whether edge alignment is possible.

As explained in Section 2.2, a depth-first search (DFS) traversal can be divided into two main phases: *discovery* and *backtracking*. During the testing phase, this mechanism is utilized to manage back edges and systematically resolve constraints. As each edge is discovered, all outgoing back edges are added to the right side of the stack in an interval with their own. Once all outgoing edges of a vertex have been processed, the algorithm backtracks to the parent edge, which is a tree edge, and addresses all constraints for edges located above the *stack_bottom* of the tree edge i.e., for the outgoing edges of the current vertex.

Resolving constraints consists of three fundamental operations, which will be discussed in detail later:

1. **Merging**: Back edges with a *same-constraint* are grouped into a single interval, reducing the number of stack units.
2. **Conflict Handling**: Back edges with a *different-constraint* are assigned to a conflict pair, represented as the left and right intervals of the same stack unit.
3. **Trimming**: Back edges with higher lowpoints are removed from the stack, as they are no longer relevant for future constraints.

If merging or conflict handling cannot be completed due to non-resolvable constraints, the algorithm terminates and concludes that the graph is not planar. After resolving all constraints for a tree edge, the algorithm proceeds to process any remaining outgoing edges of the source vertex. If no additional outgoing edges are present, the algorithm backtracks to the next parent edge and repeats this procedure. This process continues until the traversal reaches the root of the DFS tree. If this point is reached without termination, the graph is planar.

3.3.2 Constraint Handling

In this section, we provide more details on resolving constraints arising from a new outgoing edge, $e_i \in E^+_{1-i}(v)$, and its interaction with previously processed edges, e_1 to e_{i-1}. Resolving these constraints is a critical step in determining whether the graph is planar. The operations involved in constraint handling include merging edges with a *same-constraint*, handling conflicts for edges with a *different-constraint*, and trimming edges that are no longer relevant for future constraints. This section also provides pseudo-code and examples illustrating cases of non-planarity.

Merging is applied to edges with a *same-constraint*, meaning the edges must be placed on the same side of the graph. This operation groups multiple intervals into a single interval on one side of the stack, simplifying its structure and ensuring that same-constraints are respected.

Brandes' algorithm optimizes the merging process by not only merging edges when required by a same-constraint but also whenever it is possible without violating a different-constraint. This means that edges are proactively merged unless they are explicitly required to be on opposite sides. By doing so, the algorithm maintains a more structured stack, ensuring that edges are always positioned optimally, reducing complexity, and improving efficiency in handling constraints.

When processing an edge e_i, all return edges traversed since e_i are represented in the top conflict pairs on the stack, from the top down to (but not including) stack_bottom[e_i]. Due to the same-constraints induced by the fundamental cycle of lowpoint_edge[e_i],

these intervals must be grouped into a single interval on one side of the stack. The merging process is described in the pseudocode 3.1:

Algorithm 3.1: Merging

```
1  while there are conflict pairs above stack_bottom[e_i] do
2      Get the top conflict pair Q from the stack;
3      if the right interval of Q is empty then
4          Swap the left and right intervals;
5      end
6      if both the left and right intervals of Q are non-empty then
7          return The graph is not planar;
8      end
9      if the lowpoint of the low edge of the right interval is higher than the lowpoint
         of the parent edge e then                          // merge intervals
10         if the right interval of P is empty then
11             Create a new conflict pair P, copying Q;
12         else
13             Replace the low edge of P with the low edge of Q;
14             Store a reference from the replaced edge to the new edge in ref;
15         end
16         Place the new conflict pair P back on the stack;
17     end
18 end
```

If a conflict pair contains two non-empty intervals, merging on one side would result in a contradiction with previously established constraints. We will provide an example for this non-planarity condition later in this work. In such cases, the algorithm determines that the graph cannot satisfy the required conditions for planarity, halts execution, and concludes that the graph cannot be embedded without crossings.

The non-planarity condition explained above might not be trivial, but we will illustrate it with an example, as shown in Figure 3.1. In this situation, edge a is already on the stack as a lowpoint edge. The current top conflict pair at this point can be represented as:

$$Q = \{\text{left} : [d, e], \text{right} : [c, b]\}$$

This structure implies that edge f could potentially belong to the right interval. However, since the algorithm entered the constraint handling section of the algorithm, it can be inferred that another lowpoint edge must exist further down in the stack. This is because the algorithm does not handle constraints for lowpoint edges, and there is always at least one lowpoint edge present. Therefore, we know that edge f would intersect with the lowpoint edge, making the graph non-planar.

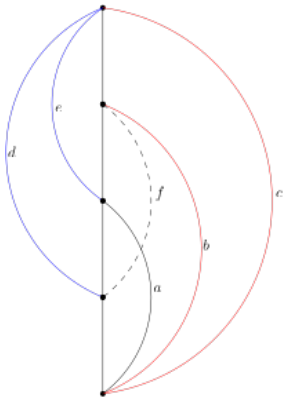


Figure 3.1: Example graph K_5 in a state where the merging operation halts due to non-planarity.

Conflict Handling is applied to resolve edges with a *different-constraint*, where edges must be placed on opposite sides of the graph. For a parent edge e, we say that the return edges of its outgoing edges $e_1, \ldots, e_{i-1}$ with lowpoints higher than lowpt$[e_i]$ are subject to pairwise same-constraints among themselves and subject to a different-constraint regarding at least one return edge of e_i, which is conflicting.

The conflict handling process iteratively resolves conflicting edges by placing them in intervals on different sides of the stack until all return edges are either merged or removed from the stack. The pseudocode 3.2 describes the steps:

Algorithm 3.2: Handling Conflicts

```
while there are conflict pairs above stack_bottom[e_1] and they are conflicting do
    Get the top conflict pair Q from the stack;
    if the right interval of Q is conflicting then
        Swap the left and right intervals;
    end
    if both the left and right intervals are conflicting then
        return The graph is not planar;
    end
    Let the low edge of the right interval of P refer to the high edge of the right
     interval of Q;
    if the low edge of the right interval of Q is defined then
        Copy it to the low edge of the right interval of P;
    end
    if the left interval of P is empty then
        Create a new left interval for P, copying the left interval of Q;
    else
        Replace the low edge and let the low edge of the left interval of P refer to
         the high edge of the left interval of Q;
    end
    Place the new conflict pair P back on the stack;
end
```

If both the left and right intervals of a conflict pair contain edges that end above lowpt$[e_i]$ and finds another conflict, the algorithm detects a contradiction with earlier constraints and halts, concluding that the graph is not planar. We will provide an example of such non-planarity later Figure 3.2.

This condition for non-planarity is illustrated in Figure 3.2. Unlike the merging step, where it is not immediately obvious that merging is prohibited when both the left and right intervals are non-empty, it is more evident that the stack cannot hold a third interval if a full conflict pair is already at the top. However, even in this part of the algorithm, the conflict pairs do not explicitly contain the lowpoint edges a and c. Nevertheless, the section is only reached if these edges were already present somewhere lower in the stack.

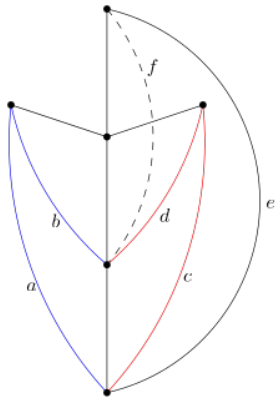


Figure 3.2: Example graph in a state where conflict handling halts due to non-planarity.

Trimming After handling the constraints for a tree edge $e \in T$, the trimming operation begins. In this step, all edges e_i in the stack are removed if lowpoint$[e_i] =$ height(source_node(e)). Edges with lowpoint$[e_i] >$ height(source_node(e)) do not need to be checked, as they would have already been trimmed earlier.

The trimming operation is divided into three phases. First, entire conflict pairs are removed if the lowest lowpoint of all edges in both intervals is equal to height(source_node(e)). Next, the left interval is trimmed by iteratively removing high edges e_h where lowpoint$[e_h] =$ height(source_node(e)), replacing them with the edge they reference until lowpoint$[e_h] <$ height(source_node(e)). This process is then repeated for the right interval.

Assigning Sides Throughout the testing phase, each edge is assigned the attributes `ref` and `side`. For back edges, these attributes are updated during the previously described stack operationsmerging, conflict handling, and trimming. Whenever an edge is removed from the stack or repositioned to the opposite side, its reference edge and side are recorded. Once all outgoing edges of a tree edge have been processed, the tree edge itself is assigned its highest return edge as its reference edge.

3.4 Embedding Phase

For the planar embedding, it is essential to determine the side for each edge and the nesting order in which edges are placed around each vertex. The order of outgoing edges is already determined during the first phase, but the order of incoming edges is computed in the third phase.

Sorting edges involves recursively computing the sign for each edge to determine its side in the planar embedding. For an edge e, if ref$[e]$ has no reference edge, the sign is directly given by side$[e]$, where -1 indicates the left side and $+1$ the right. If ref$[e]$ refers to another edge, the sign of e is calculated as the product of side$[e]$ and the sign of ref$[e]$. Since the sign of ref$[e]$ might not yet be known, it must be recursively computed until it refers to an edge without a reference edge. This ensures that all edges are consistently

placed according to the constraints resolved during the testing phase. As the side of an edge may change during testing, this step is only performed after the testing phase is complete.

To place the incoming edges around each vertex, the DFS tree is traversed once again in the order of the nesting depth multiplied by the sign of each edge. This traversal ensures that all edges on the left side are in reversed order. They will then be placed either as the most left edge on the left side or as the most right edge on the right side in the circular order around the vertex. This final placement satisfies the constraints of the planar embedding. Resulting in an *LR Ordering* defined as follows:

Definition 9 (LR Ordering)**.** Let v be a vertex with outgoing edges divided into left edges $\{e_{L_1}, \ldots, e_{L_\ell}\}$ and right edges $\{e_{R_1}, \ldots, e_{R_r}\}$ by the LR partition. If v is not the root, let (u, v) be its parent edge. The clockwise LR ordering of the edges around v is:

$$(u,v), L(e_{L_\ell}), e_{L_\ell}, R(e_{L_\ell}), \ldots, L(e_{R_1}), e_{R_1}, R(e_{R_1})$$

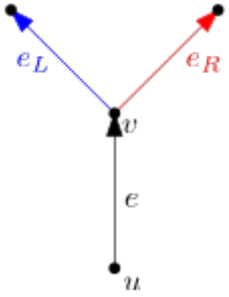


Figure 3.3: Graph showing the order of outgoing edges around vertex v.

Here, (u, v) is omitted if v is the root. $L(e)$ and $R(e)$ represent the incoming back edges associated with cycles on the left and right of e.

Within $L(e)$ or $R(e)$, the order of back edges is based on their corresponding cycles. For two back edges $b_1 = (x_1, v)$ and $b_2 = (x_2, v)$ in $R(e)$, b_1 precedes b_2 if the fork of their cycles determines $(x_1, y_1) \prec (x_2, y_2)$. For back edges in $L(e)$, the order is reversed.

CHAPTER 4

Literature Review

As outlined earlier, this thesis describes the algorithm based on the work of Brandes [Bra09], which itself is a refinement and simplification of the algorithm originally proposed by de Fraysseix, de Mendez and Rosenstiehl in multiple versions [dFdMR06][dF08][dFdM12]. In the following chapter, we refer to this paper with FMR. Similarly, Brandes' work will occasionally be referred to as B for brevity. In this chapter, the focus will be on identifying and analyzing the differences between the two publications, with particular attention to distinguishing between genuine algorithmic modifications and changes that are merely stylistic or aimed at enhancing clarity without altering the underlying logic.

One key distinction between the two works lies in their use of terminology. Brandes frequently employs synonyms or slightly modified terms for concepts that are structurally identical to those introduced by FMR. This will be represented in Section 4.1, where equivalent terms are systematically listed and explained. Additionally, Brandes provides more comprehensive explanations for certain definitions, often rewriting entire concepts in a way that emphasizes accessibility and practical application. This will be analyzed in Section 4.2, where differences in phrasing and elaboration are critically examined. These differences contribute to a perception of divergence between the works, even when the foundational ideas remain consistent.

The most significant differences emerge in the algorithm itself. This discrepancy arises primarily because FMR presented the algorithm in a concise and abstract manner, while Brandes provides a detailed, step-by-step explanation that integrates previously defined concepts into a cohesive and practical framework. These differences will be analyzed in Section 4.3, where the structure, data handling, and procedural steps of the two algorithms are compared.

Between 2006 and 2012, three papers titled *Trémaux Trees and Planarity* were published, expanding upon the work of FMR. The first paper, published in 2006, lists Rosenstiehl as a co-author, whereas the subsequent versions, published in 2008 and 2012, acknowledge his

involvement only in initiating the research. Each version introduced minor improvements, including additional proofs, examples, and clearer illustrations. Furthermore, the later versions added two chapters, *How to Draw a Map on the Plane* and *Trémaux Trees and Kuratowski Subdivisions*. However, these chapters are not part of the focus of this thesis and are therefore excluded from further consideration. Brandes work is explicitly based on the original 2006 publication, but this thesis also takes into account the updated versions, providing a broader perspective on the evolution of the algorithm.

4.1 Glossary

This glossary shows synonyms used in the works of Brandes and FMR, highlighting equivalent terms and their definitions.

Table 4.1: Glossary of Synonyms

Definition	**Brandes**	**FMR**
A spanning tree derived from a Depth-First Search (DFS) traversal. It represents the hierarchical structure of the graph.	DFS tree	Trémaux tree
Non-tree edges in a DFS traversal. These edges connect a vertex to one of its ancestors, forming cycles.	Back edge	Cotree edge
The lowpoint of an edge is its lowest return point, if any, or its source if none exists.	Lowpoint	Low
A cycle formed when a cotree edge is added to a Trémaux tree, fundamental for determining planarity.	Fundamental Cycle	Circuit (Fundamental Cycle is also used)
Partition of back edges into two sets (left and right) to prevent crossings in a planar embedding. The partition is based on their fundamental cycle orientations.	LR Partition	F-coloring
A refined partitioning method that resolves ambiguities in edge placement. It ensures all edges sharing the same lowpoint are aligned in the same set.	Aligned LR Partition	Strong F-coloring
Circular order of edges around each vertex. This order is derived from edge partitions and guides the planar embedding process.	LR ordering	σ-map

A partial order defined on edges to represent their relative hierarchical nesting in the graph. It is used to guide embedding or traversal.	Nesting-order	Low-order

4.2 Comparison of similar terms and concepts

To better understand the similarities and differences between the terminologies and concepts used by FMR and Brandes, this section provides a detailed comparison of key terms, highlighting how their definitions align or diverge and their impact on the respective algorithms.

Return edge (B) – Fringe (FMR) Both *fringe* and *return edge* describe interactions between non-tree edges and the tree structure, but they differ in scope:

- *Fringe*: The fringe of a tree edge $e = (x, y)$ is the set of all cotree edges that connect a vertex in the subtree rooted at y to a vertex strictly smaller than x in the Trémaux tree. For a cotree edge f, its fringe is simply $\{f\}$. The fringe helps classify tree edges into types such as block, thin, or thick edges, as defined later, based on the lowpoints of the edges in the fringe. An example is given in Figure 4.1.
- *Return Edge*: Recalling the definition from above, a return edge is a back edge that leads to a return point, which is a vertex located strictly lower in the DFS tree than the source of the tree edge. A back edge is a return edge for at least one tree edge.

Key Difference: Fringe provides a collective view of all return edges tied to a tree edge, while a return edge focuses on the connection between a single back edge in relation to a tree edge. For any tree edge e, the set of all its return edges forms the fringe of e. Both concepts describe the same interaction between back edges and tree edges, but are used in slightly different context.

Chordal edge (B) – Thick edge (FMR) Both *chordal edge* and *thick edge* are characteristics of tree edges in a graph, but they are defined differently.

- *Chordal Edge*: Recalling the definition from Brandes, a chordal edge is a tree edge with multiple return edges.
- *Thick Edge*: In FMR a thick edge is defined as a tree edge that is neither thin nor blocked. A *blocked edge* is a tree edge without return points, while a *thin edge* has exactly one return point. Meaning, a thick edge is an edge with multiple return points. An example is given in Figure 4.1.

Key Difference: While chordal edges must have multiple return edges, an edge is only defined as thick if it has multiple return points. Therefore, an edge with multiple return edges to the same return point is chordal but not thick. Both concepts are used to define the order in which edges are processed in the DFS traversal. This small difference in definition might alter the processing order between the two algorithms.

Stem (FMR) The concept of *stem* is explicitly defined in FMR but is not directly named in Brandes' work. Instead, its functionality is used implicitly in the algorithm.

- *Stem*: In FMR, the stem function is defined on a vertex v and an edge e that are both part of the same oriented fundamental cycle. The stem is the first edge after v and before e in the cycle orientation, or e itself if it is directly after v. The stem function is primarily used to compare two edges from different cycles by applying the function to their last common overlapping vertex. An example is given in Figure 4.1.
- *Implicit Use by Brandes*: While Brandes does not explicitly define or refer to the stem function, the concept is implicitly used in the processing of forks. Recalling the definition of a fork, it is defined by a branching point v with its incoming tree edge e and two outgoing edges e_1 and e_2. These edges lead into two cycles, c_1 and c_2. All edges in c_1 have e_1 as their stem when considered in the context of branching point v, and all edges in c_2 have e_2 as their stem. This implicit use of the stem function helps guide the traversal and processing of cycles in the algorithm.

Key Observation: The stem function in FMR is explicitly formalized for comparing edges across different cycles, while in Brandes, it is implicitly used as part of the logic for handling forks and traversing fundamental cycles. This difference highlights the contrasting levels of detail in the two descriptions.

Fringe-opposed subset (FMR) The concept of the *fringe-opposed subset*, often referred to as *fop*, is explicitly defined in FMR but is implicitly utilized in Brandes' algorithm through the context of forks.

- *Fop*: In FMR, the fringe-opposed subset $\mathrm{Fop}_{e_2}(e_1)$ is defined for a vertex v and two outgoing edges $e_1, e_2 \in E^+(v)$. It is given by:

  $$\mathrm{Fop}_{e_2}(e_1) = \{f \in \mathrm{Fringe}(e_1) : \mathrm{low}(f) \succ \mathrm{low}(e_2)\}.$$

  The fringe-opposed subset captures the subset of fringe edges associated with e_1 that have a lowpoint greater than that of e_2. An example is given in Figure 4.1.
- *Implicit Use by Brandes*: Similar to the stem, this concept also uses the framework of a fork when analyzed in detail. When comparing two outgoing edges of a fork, the fringe-opposed subset corresponds to all the return edges associated with one of the outgoing edges of the fork.

Key Observation: Both concepts play a central role in the definition of the LR planarity criterion, which will be shown in the next paragraph.

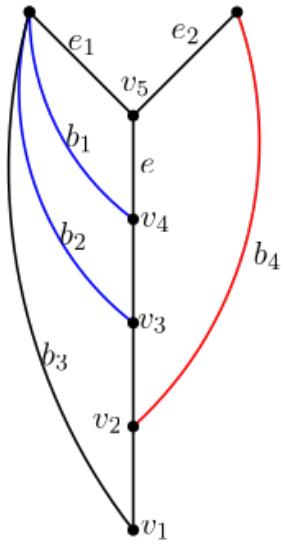


Figure 4.1: Example graph (from Section 2.4) illustrating key terms. See details below.

The figure illustrates the following terms:

- *Fringe*, e.g., $\text{Fringe}(e_1) = \{b_1, b_2, b_3\}$.
- *Thick Edge*, e.g., e_1.
- *Thin Edge*, e.g., e_2.
- *Stem*, e.g., $\text{Stem}(v_5, b_1) = e_1$.
- *Fop*, e.g., $\text{Fop}_{e_2}(e_1) = \{b_1, b_2\}$.

F-Coloring (FMR) – LR Partition (B) *F-coloring* and *LR Partition* refer to the same underlying concept of dividing all back edges into two sets. Although the terminology differs significantly, a closer examination reveals that both describe the same fundamental idea.

- *F-Coloring*: A coloring $\lambda : E \setminus T \to \{-1, 1\}$ of the cotree edges, such that for every vertex v, the fringe-opposed subsets $\text{Fop}_{e_2}(e_1)$ and $\text{Fop}_{e_1}(e_2)$ are monochromatic and colored differently.
- *LR Partition*: A partition $B = L \cup R$ of back edges into two classes, left (L) and right (R), such that for every fork with outgoing edges e_1 and e_2:

1. All return edges of e_1 that end strictly higher than lowpt(e_2) belong to one class.
2. All return edges of e_2 that end strictly higher than lowpt(e_1) belong to the other class.

Key Observations:

- The group of edges referred to in Brandes' criterion 1, "All return edges of e_1 that end strictly higher than lowpt(e_2)", is equivalent to $\text{Fop}_{e_2}(e_1)$ in FMR's framework. Similarly, the edges referred to in criterion 2 correspond to $\text{Fop}_{e_1}(e_2)$.
- In Brandes' LR Partition, all edges in one group must belong to the same class, meaning that they must be monochromatic. Furthermore, the two groups must belong to different classes, which aligns exactly with the condition in F-coloring that the groups are colored differently.
- Classification labels: FMR uses two colors (-1 and 1), while Brandes uses left (L) and right (R) classes. These different labels do not affect the algorithm and are therefore technically equivalent.
- While FMR states that the criterion must be applied for every vertex, Brandes applies it specifically to every fork. This distinction is superficial, as a vertex without two outgoing edges does not have a fringe-opposed subset, satisfying the criterion automatically. Realizing this, the definitions are identical.

Both concepts are the core criterion for determining whether a graph is planar. If the criteria can be satisfied for all edges, the graph is planar; otherwise, it is not.

4.3 Algorithm Comparison

This section focuses on comparing the algorithms provided by FMR and by Brandes. The analysis will examine differences in each step and the underlying data structures referenced in the respective pseudocodes.

It is important to note that this analysis is based solely on the provided pseudo-code. Brandes pseudo-code is detailed and straightforward to translate into a practical implementation, with every step clearly explained to facilitate direct application. In contrast, the pseudo-code by Rosenstiehl and de Fraysseix is more abstract and high-level, outlining the main steps but not providing pseudo-code for critical aspects such as constraint handling, which is central to the planarity test. To better understand constraint handling, relevant concepts are inferred from other sections of the paper where applicable.

Due to this disparity, some steps from FMR cannot be compared in detail with Brandes' algorithm. However, the underlying logic, structures, and methodologies will be analyzed whenever possible to provide insight into how the two algorithms align and differ.

Main data structure Both algorithms rely on stacks to manage the back edges during the planarity test, but differ in their initialization and organization.

Brandes' algorithm uses a single stack for all back edges, adding edges one at a time as they are processed. This centralized approach simplifies the used data structure from the beginning.

In contrast, FMR initialize a separate stack for each back edge. As the algorithm progresses, the stacks for outgoing edges of a tree edge are merged into the stack of the parent edge. This merging process continues down the Trémaux tree until a single stack remains.

Although the final result is a similar data structure, the two approaches differ in how they build and manage their stacks during the traversal.

Stack Units Recall that stack units are conflict pairs of edges, which is a concept used by both algorithms. However, the way these conflict pairs are represented in the stacks differs between the two approaches.

In FMR's algorithm, the stack directly stores individual edges as conflict pairs without any grouping or interval-based representation.

In Brandes' algorithm, the conflict pair units are intervals of edges. When a new edge is added, it initially represents its own interval. However, during the processing of a parent edge, edges belonging to the same same-constraint are grouped into a single interval, represented only by its highest and lowest edges. All intermediate edges are stored in the `ref` array. This is possible because no edge within the interval will independently belong to another constraint. Even if the entire interval belongs to a different-constraint, this grouping remains valid.

This interval-based approach reduces the storage requirements of the stack and simplifies certain computations by minimizing the number of elements that need to be processed directly.

Storing Edge Attributes Both algorithms collect various properties, such as lowpoint, lowpoint edges, thick/chordal classification, and others, during their execution. However, they differ in how and where these attributes are stored.

In FMR's approach, most attributes are stored at the vertices rather than directly on the edges. This is based on the observation that, except for the root, every vertex in a DFS spanning tree (ignoring back edges) belongs to exactly one tree edge. By contrast, Brandes' algorithm stores all attributes directly on the edges.

Storing attributes on the edges, as in Brandes' algorithm, has the advantage of making the data more immediately accessible during edge processing, where the information is typically needed. However, FMRs approach of storing attributes on vertices can be more storage efficient, as the number of vertices in a graph is generally smaller than the number of edges, leading to more compact data structures.

The choice between these approaches reflects a tradeoff between accessibility and storage efficiency, which may vary in significance depending on the graph's structure and the specific algorithmic requirements.

Handling Lowpoint Edges Both algorithms store the lowpoint edge for each tree edge, as it is essential for evaluating the planarity criterion. However, their approaches to handling lowpoint edges differ.

In FMR's algorithm, lowpoint edges are processed in the same way as all other back edges. By contrast, Brandes' algorithm does not handle constraints of lowpoint edges, as they are not subject to any constraints at the time they are added to the stack. Instead, they may become relevant for edges added later, at which point any necessary checks will still be performed when processing those edges. This approach improves the efficiency of the algorithm by reducing unnecessary computations.

Trimming Both algorithms perform trimming by removing processed edges from the stack if their return point is not lower than the source of all processed edges, as such edges will not be subject to future constraints.

In FMR's algorithm, trimming is relatively straightforward since edges are stored directly on the stack. In Brandes' algorithm, however, trimming can be more complex because edges may be part of an interval rather than stored individually on the stack. This requires checking the references within the interval to identify and trim the appropriate edges accurately.

Traversal Order Both algorithms traverse the DFS tree for the second and third passes based on the lowpoints of each edge. In the third traversal, edges on the left side are processed in reverse order.

Brandes achieves this efficiently by adjusting the traversal order with a simple multiplication: the traversal direction is determined by the side multiplied with nesting depth, where the left side is assigned -1 and the right side is assigned 1. This straightforward approach ensures that the left side is automatically processed in reverse order.

In contrast, FMR handle the reversal more elaborately by creating a new order, called the "out-order". This order determines the side for each edge again, using criteria already established in the second traversal step. While functional, this approach introduces additional complexity compared to Brandes' simpler method.

Embedding In FMR, the embedding step is not described in detail; it is only mentioned that the embedding is derived from historical data. As a result, a direct comparison of this step is not possible. However, it is reasonable to assume that the embedding procedure follows a similar approach to the one outlined by Brandes.

Summary

This chapter has analyzed the similarities and differences between the works of Brandes (B) and de Fraysseix, de Mendez, and Rosenstiehl (FMR). The analysis spans terminology, concept definitions, and algorithmic implementation. Below is a summary of the key observations:

1. **Glossary of Synonyms:** A systematic glossary was provided, highlighting synonymous terms and concepts between Brandes and FMR. Despite differences in phrasing, most terms describe the same fundamental ideas, with slight variations in usage or context.

2. **Planarity Criterion:** The definitions and concepts from both works converge in the planarity criterion. Throughout both papers, the foundational ideas build upon one another until the criterion can be formally expressed, demonstrating the shared logic underlying their approaches.

3. **Standalone vs. Dependent Definitions:** Brandes focuses on definitions that can stand independently, making them more accessible and straightforward. Conversely, FMR introduces dependent functions and definitions that rely on prior ones, which can make the concepts harder to grasp without resolving earlier references.

4. **Algorithm Representation:** Most of the differences in the algorithms are related to representation, readability, and memory efficiency rather than the algorithms functionality. These adjustments aim to either improve clarity or streamline resource usage, without affecting the overall logic.

5. **Interval Usage in the Stack:** The primary algorithmic difference lies in Brandes use of intervals in the stack. This approach reduces storage requirements and simplifies certain operations by consolidating edges into intervals. However, it also introduces additional computational effort in some cases, as intermediate edges that are part of the interval must be retrieved from references when needed in other operations.

CHAPTER 5

Implementation Details

A significant focus of this thesis is the implementation of the planarity testing algorithm based on the pseudocode provided by Brandes [Bra09]. In later chapters, this implementation will be evaluated for efficiency and compared against other state-of-the-art planarity testing algorithms.

This chapter provides an overview of the implementation, highlighting key aspects and adjustments made to the original pseudocode. These adaptations aim to ensure practical applicability while adhering to the algorithm's theoretical framework.

Following this, we will analyze differences between our implementation and the 2003 implementation by de Mendez and de Fraysseix in PIGALE [dFdM02]. The analysis will focus on differences in data structures and whether these differences represent genuine algorithmic changes or are merely refactorings aimed at improving clarity or efficiency.

5.1 Our Brandes Implementation

This section focuses on implementing the planarity testing algorithm from Brandes' pseudocode [Bra09]. It includes an overview of the chosen tools and techniques, as well as a discussion of key adaptations made to the pseudocode.

5.1.1 Implementation Details

The algorithm was implemented in C++, a language particularly suited for graph algorithms due to its low-level memory management capabilities and support for efficient computation. Its ability to directly control system resources ensures that complex algorithms can run with minimal overhead. Moreover, using C++ allows for a direct and fair comparison with other planarity testing implementations, such as the PIGALE library.

The implementation leverages the Open Graph Drawing Framework (OGDF) [CGJ+13], a specialized C++ library designed to bridge the gap between theory and practice in graph drawing. OGDF offers a wide variety of algorithms and data structures, including support for planarity testing and graph decomposition. Its modular and reusable architecture provides a robust platform for developing and extending graph-related algorithms, making it an ideal choice for this implementation.

To ensure correctness, assertions were added at various stages of the algorithm. These checks include a critical assertion that verifies the validity of the embedding when the graph is determined to be planar, ensuring the integrity of the implementation and the accuracy of the results.

5.1.2 Adaptations to the Pseudocode

The following section outlines key adaptations made during the implementation of Brandes' pseudocode. These changes were introduced to address potential limitations, improve efficiency, or resolve bugs, ensuring the algorithm is both robust and performant.

Iterative DFS Implementation All three DFS runs in the algorithm were implemented iteratively instead of recursively. This approach uses an explicit call stack where each stack element corresponds to an edge. Each element contains two phases: discovery and backtracking. During discovery, a vertex is visited, and its neighbors are processed; during backtracking, the algorithm revisits the element to handle post-discovery operations. Once the backtracking phase is complete, the element is removed from the stack. This iterative process continues until the stack is empty.

The iterative approach was chosen to prevent stack overflow, which can occur in recursive implementations for graphs with a large depth. In C++, the recursion stack is limited by the system and may lead to runtime errors if the depth of the recursion exceeds this limit. By using an explicit call stack, the implementation avoids this limitation while maintaining the correctness of the algorithm.

One-Dimensional Array for Adjacent Edges Instead of utilizing a two-dimensional array to store the adjacent edges for each node, as is commonly implied in graph algorithms, a one-dimensional array was employed. Each node is assigned a contiguous range in the array, defined by a starting and ending index. This adaptation reduces memory movements, as it avoids allocating and deallocating multiple small arrays for each node. By consolidating the data into a single array, the algorithm benefits from better cache locality and reduced overhead, leading to improved performance.

Index-Based Stack Bottom Tracking During the construction of the constraint stack, it is necessary to track the `stack_bottom` for each edge. Instead of storing the actual stack element, the implementation stores an integer index that refers to the element. This approach is more memory-efficient and avoids potential issues caused by

modifications to stack elements. The integer index remains valid regardless of changes to the stack, ensuring accurate and reliable access to the required elements.

Handling Lowpoint Edges Referencing Themselves In some cases, lowpoint edges may reference themselves. While this does not affect planarity testing, it can lead to errors during the embedding phase. Since lowpoint edges store their side directly, and not via a reference to another edge, the implementation bypasses the reference entirely and directly uses the stored side. This ensures correct behavior during the embedding phase and resolves potential errors arising from self-referencing lowpoint edges.

Fix for Lowpoint Edge Comparisons In the original pseudocode, lowpoint edges were only checked against the lowpoint edges of parent edges, overlooking other edges within the same fundamental cycle induced by the lowpoint edge. This oversight could lead to incorrect references in cases where another edge in the cycle already had a lowpoint edge that should serve as a reference. The implementation corrects this by ensuring that all edges within the fundamental cycle are considered, allowing proper references to be established and ensuring correctness in handling such cases.

Alternative Embedding Phase Implementations The embedding phase in the pseudocode involves inserting edges into the adjacency array, as proposed by Brandes. While functional, this approach can be inefficient due to the search algorithm used for locating the correct position for insertion, which is not guaranteed to run in linear time. To address this, an alternative implementation was developed using a doubly linked list to store adjacent edges for each node. This structure allows for efficient insertion operations, ensuring linear time complexity. The use of this workaround, originally suggested by Brandes, improves the overall performance of the embedding phase without altering the algorithm's correctness.

5.2 The PIGALE Implementation

In this section, we compare our implementation of Brandes' pseudocode [Bra09] with the planarity testing algorithms implemented in PIGALE [dFdM02]. PIGALE provides two planarity testing methods: *newPlanarity*, a more recent and optimized approach, and *planarity*, the original implementation. For this comparison, we focus on the *planarity* method, as it provides a more direct basis for understanding the conceptual and structural differences between the two algorithms. The newer *newPlanarity* method will be evaluated later in the context of efficiency tests, given its claimed performance improvements.

This analysis examines various aspects of the implementations, including the employed data structures, algorithmic design, overall structure, and the frameworks used. By comparing these elements, we aim to highlight the differences and similarities between the two approaches, providing a deeper understanding of the evolution of planarity testing techniques.

5.2.1 Comparison of Frameworks: PIGALE vs. OGDF

One of the major differences between the two implementations lies in the frameworks used. PIGALE is a lightweight and specialized library, focusing solely on graph algorithms and data structures, with minimal overhead. This simplicity enables efficient data handling, as the framework is tailored for fast computation and streamlined memory usage [Tam13]. On the other hand, OGDF is a comprehensive framework designed to bridge the gap between theory and practice in graph drawing and related algorithms. While OGDF offers a rich set of functionalities and reusable components, its modularity and generality can introduce additional computational overhead compared to a more focused framework like PIGALE [CGJ+13].

A key difference between the two frameworks is their approach to graph data structures. OGDF is designed as a dynamic library, allowing for modifications such as adding or removing edges and nodes after a graph has been created. This flexibility makes it well-suited for applications requiring iterative modifications or real-time updates. In contrast, PIGALE employs a static structure, meaning that once a graph is constructed, it cannot be modified directly. Instead, any structural change, such as adding an edge, requires reconstructing the entire graph. This design choice reflects PIGALEs focus on planarity testing, where dynamic updates are unnecessary, and optimizing for computational efficiency is more important than graph mutability.

The choice of framework has a direct impact on the implementation strategies. PIGALEs simplicity facilitates low-level optimizations, making it well-suited for implementing highly efficient algorithms. OGDF, by contrast, provides advanced abstractions and pre-built structures that simplify development but may involve trade-offs in terms of raw performance. This distinction underscores the differing design philosophies of the two implementations and their implications for planarity testing.

5.2.2 Orientation Phase

Three Phases vs. Unified Approach In PIGALE, the orientation phase is divided into three distinct subphases:

- **DFSRun**: Constructs the DFS tree.
- **bicon**: Classifies edges as thick, thin, or leaf edges, where leaf edges connect to another biconnected component.
- **LRSort**: Establishes an ordering for the subsequent testing phase.

It is important to note that only the first step involves a DFS traversal; the subsequent steps merely iterate through the edges, leveraging the structure already established by the DFS tree. By contrast, our implementation integrates all these tasks into a single DFS traversal. During this traversal, the DFS tree is built, edge types are classified (e.g.,

chordal edges are identified), and the nesting depth for each edge is calculated. The nesting depth serves as the metric for ordering edges. This unified approach avoids the need for multiple iterations through the edge list, reducing computational overhead and improving efficiency.

DFS Tree Construction and the Edge Object In PIGALE, the DFS tree is constructed during the first step of the orientation phase using the `nvin` data structure. This compact array efficiently maps edge indices to their corresponding endpoints:

- For an edge e_i, `nvin[i]` represents the target node, while `nvin[-i]` represents the source node.
- Indices from 1 to $n-1$ correspond to tree edges (where n is the number of vertices).
- Indices from n to m represent back edges (where m is the total number of edges).

This structure simplifies edge classification and allows direct access to edge endpoints without requiring additional data structures. Furthermore, the contiguous memory allocation of the `nvin` array ensures fast indexing and minimal memory movement.

In contrast, our implementation represents edges as objects with multiple attributes, such as source, target, type, and references to associated edges. This approach provides greater flexibility and modularity, enabling the algorithm to extend edge functionality easily. However, it also incurs additional memory overhead and slower access times compared to PIGALEs streamlined `nvin` structure.

To ensure a fair comparison, the DFS tree generated by PIGALEs first step (`DFSRun`) was used as input for our implementation. Although the DFS tree constructed by PIGALE may differ from the one generated by our algorithm, this discrepancy does not affect the planarity result. However, it can influence the processing order, potentially leading to variations in intermediate operations. By using the same initial DFS tree, both implementations were aligned for a more meaningful comparison.

Edge Classification PIGALE uses three classifications for edges during the `bicon` phase:

- **Thick edges:** Edges with multiple return edges.
- **Thin edges:** Edges with exactly one return edge.
- **Leaf edges:** Edges that connect to another biconnected component.

Edges are stored in separate stacks based on these classifications, facilitating their retrieval and processing in later phases.

In our implementation, edge classification is simplified by focusing on whether an edge is chordal (i.e., it has multiple return edges). This approach eliminates the need for separate stacks for edge types, as the classification directly influences the processing order. By streamlining classification, we reduce the overhead associated with maintaining multiple stacks while preserving the structural information necessary for planarity testing.

Ordering Mechanism The *LRSort* phase in PIGALE determines the order in which edges are processed during the testing phase. This mechanism relies on three auxiliary arrays:

- **tref:** Stores the first outgoing tree edge for each node.
- **tel:** Stores the first outgoing edge leading to the lowpoint.
- **linkt:** A linked list connecting edges to define their order.

For example, the order of edges for a node n_i can be determined as follows: Start with `tref[i]` to retrieve the first edge. Then, follow the links: `next = tel[i]`, `next = linkt[next]`, and so on until `next == 0`. This mechanism ensures that edges are processed in the correct order based on their classification and structural relationships.

In our implementation, the same ordering is achieved by calculating the *nesting depth* during the DFS traversal. Edges are then sorted based on their nesting depth, which reflects their hierarchical relationships in the DFS tree. However, in PIGALE, this mechanism allows different types of edges (e.g., tree edges or lowpoint edges) to be handled differently without requiring additional lookups, as the classification is inherently tied to the data structure.

Discovering Back Edges and Handling Lowpoints In the PIGALE implementation, back edges are not added to the `nvin` data structure during the discovery phase. Instead, they are incorporated during the backtracking phase when the DFS traversal visits their lower endpoint. This design ensures that back edges can be inserted into the `nvin` array in reverse order, directly sorted by their lowpoints. By maintaining this sorted structure, the implementation facilitates efficient iteration over back edges in subsequent phases. This method leverages the natural ordering of back edges to reduce computational overhead in processing steps that rely on their lowpoints.

Minor Differences Several minor differences between the PIGALE implementation and our approach can be observed:

- **Edge and Node Labels:** PIGALE starts indexing edges and nodes at 1, while our implementation begins at 0.

- **Node Heights:** PIGALE uses the term *dfsnum* to refer to node heights, while our implementation explicitly uses the term *height*.
- **First Outgoing Tree Edge:** In PIGALE, the first outgoing tree edge of a node always has the same index as the node itself. This indexing is leveraged to store the biconnected status of the edge (thick, thin, or leaf).
- **Chordal vs. Thick Edges:** In our implementation, an edge with multiple return edges to the same lowpoint is classified as *chordal*. In contrast, PIGALE does not classify such edges as *thick*.
- **Handling Non-Biconnected Components:** PIGALE processes non-biconnected components separately, while our implementation integrates this handling directly into the main algorithm.

5.2.3 Testing phase

DFS Traversal In the testing phase, the DFS traversal in PIGALE follows a specific order based on the data structures *tref*, *tel*, and *linkt*. The traversal starts by following tree paths, using the reference edge stored in *tref*. Once no further tree edge is available, the traversal continues with the lowpoint edge stored in *tel*. Subsequently, the remaining edges are processed in the order defined by *linkt*. If any edge encountered during this phase is a tree edge, the traversal resumes along the corresponding tree path as long as possible.

In contrast, our implementation employs a traversal order based on the nesting depth. This approach ensures that the lowpoint edge is always processed first, followed by edges in descending order of their nesting depth. This distinction in traversal order reflects a fundamental difference in how the two implementations prioritize edges during the testing phase.

Constraint Stack The data structure for the constraint stack in PIGALE closely resembles that of our implementation. It uses a stack where each element is a conflict pair, consisting of intervals defined by their high and low edges, referred to as *top* and *bottom*, respectively.

One notable difference is the default side for new conflict pairs. In PIGALE, the default side is set to *left*, whereas in our implementation, it defaults to *right*. Another distinction lies in the handling of stack elements when they are removed. PIGALE employs a pointer to track the current top element. When an element is "removed" from the stack, the pointer simply moves to the next element below without actually deleting the previous element.

Both of these differences are structural and do not impact the algorithm's behavior or its final results.

Stack Operations and Evolution

This subsection delves into the specific operations performed on the constraint stack, including merging intervals, deleting elements, and handling conflicts. It also highlights how the evolution of the stack differs between the PIGALE implementation and our implementation of Brandes' algorithm.

Handling Lowpoint Edges One key difference between the two implementations lies in how lowpoint edges are handled. In Brandes' algorithm, edges are traversed strictly in the order of the nesting depth, meaning lowpoint edges are visited first and placed onto the stack before other edges. This ordering simplifies many operations. For instance, when merging intervals or adding conflict pairs, it is already guaranteed that a lowpoint edge exists below in the stack. This ensures that certain conditions, such as those required for these operations, are automatically fulfilled by when entering the merging step, avoiding additional checks.

Also, Brandes extends the alignment of edges as defined in 8 by not only aligning return edges that return to the same lowpoint as the lowpoint edge, but also aligning lowpoint edges with lowpoint edges in the stack below.

In contrast, the PIGALE implementation treats lowpoint edges like all other back edges, handling them in the same order during traversal. As a result, when merging intervals or forming conflict pairs, conditions involving lowpoint edges may need to be explicitly verified. This can necessitate retrieving the lowpoint edge again to ensure all requirements are met, leading to additional computations during the execution of the algorithm.

Handling Forks The two implementations diverge in how they handle forks during the testing phase. In PIGALE, explicit pointers are maintained for the last fork in the stack. This ensures that conflicts induced by the fork can be efficiently resolved, and edges are placed accordingly. Storing a reference to the fork is particularly important in PIGALE because constraints are resolved immediately whenever a new edge is added to the stack. And they might need to be handled differently, if they are induced by a fork.

In contrast, Brandes' implementation processes constraints differently. Constraints are resolved for all elements in the stack above the stack bottom of the current edge. For instance, when backtracking to handle the constraints for a tree edge inducing a fork, all relevant constraints are already located higher up in the stack. These constraints are resolved collectively, ensuring that either the conflicts are handled, or the graph is determined to be non-planar. This approach eliminates the need for explicit fork references, as the structure of the stack inherently captures the required information.

Deleting Edges The operation of deleting edges, referred to as trimming in Brandes' algorithm, functions in an essentially identical manner in both implementations. In both cases, this operation involves removing either the entire stack element or trimming the left or right interval within the stack.

However, a key difference lies in the timing of this operation. In PIGALE, trimming occurs at the beginning of processing an edge and removes all edges with low values at the source node of the currently processed edge. By contrast, Brandes' algorithm performs trimming at the end of processing an edge. This approach removes all edges with low values at the source node of the parent edge of the current edge. While the timing differs, the underlying functionality of the operation remains consistent across the two implementations.

Merging Edges In the PIGALE implementation, edges are merged whenever a group of edges belonging to a same-constraint is interrupted by an edge with a different-constraint. This approach aligns closely with the theoretical description of planarity testing, ensuring that constraints are resolved in real-time as the edges are processed.

By contrast, Brandes' algorithm takes a different approach by merging edges only during the backtracking phase and whenever possible. This method simplifies the embedding of edges into the stack at a later stage because all edges are already positioned optimally within the stack. Additionally, if no further placement is possible during backtracking, the algorithm can immediately conclude that the graph is non-planar without needing to evaluate additional conditions. This deferred merging strategy streamlines the resolution of constraints and reduces computational overhead in subsequent steps.

Additional Operations in PIGALE The PIGALE implementation includes several additional operations that we will not discuss in detail. These derive primarily from the differences in handling lowpoint edges and the explicit storage of forks, rather than relying on a more generic approach. These operations are specific to the structural choices made in PIGALE and are unnecessary in Brandes' implementation due to its unified and streamlined handling of constraints.

5.2.4 Embedding Phase

The embedding phase in both implementations follows a similar approach, with only minor differences in the data structures used. These variations primarily affect how edges are stored and accessed during the embedding process, but the overall logic and goals remain consistent across both implementations.

HIST Data Structure The PIGALE implementation uses the *HIST* data structure to store information from the testing phase needed for reconstructing the embedding. This is analogous to the *ref* and *side* arrays in Brandes' algorithm. Brandes' implementation consolidates information about edge orientationwhether edges are on the same or opposite sides, or designated as left or right based on a reference edgeinto a single *side* array. In contrast, the PIGALE implementation divides this information into two separate arrays, *dus* and *flip*, adopting a distinct approach to managing edge orientations.

Edge Labels As discussed in *DFS Tree Construction and the Edge Object*, the DFS tree used during the testing phase is relabeled to facilitate processing. This relabeling ensures efficient traversal and manipulation of edges during the planarity testing and embedding phases. However, these changes are reversed during the embedding phase, ensuring that the output graph retains the original edge labels from the input graph. This approach maintains consistency between the input and output representations while enabling internal optimizations during the algorithm's execution.

CIR Data Structure The *cir* data structure efficiently organizes edges around each node in the graph. Implemented as a one-dimensional array, it functions like a circular linked list. Each edge is represented by an index and directly points to the next edge in the sequence, ultimately forming a closed loop. Since edges in PIGALE are stored as integers, they can serve as both values and indices, simplifying the structure. Each node forms its own closed loop within the array, providing an efficient way to manage edge references. This approach optimizes memory usage by requiring a single array with exactly the number of edges in the graph, eliminating the need for a two-dimensional array per node.

5.2.5 Summary of Implementation Comparison

This section compared our implementation of Brandes' pseudocode with the *planarity* method implemented in PIGALE. While both implementations aim to solve the same problem, they differ significantly in terms of frameworks, data structures, and algorithmic design. Below are the key differences and insights derived from the comparison:

- **Frameworks:** PIGALE is a compact and specialized library optimized for graph algorithms, designed for minimal memory usage and fast performance. It is particularly efficient for static graphs, where the structure remains unchanged during computations. In contrast, OGDF is a versatile framework focused on graph drawing and related tasks, offering greater modularity. While OGDF is more adaptable to dynamic graphs, where modifications like insertions and deletions occur, this flexibility can introduce additional computational overhead.

- **Data Structures:** PIGALE employs compact and efficient data structures, such as the `nvin` array for edge representation and the `cir` array for circular linked lists of edges around nodes. These structures minimize memory usage and facilitate fast access. Our implementation, on the other hand, uses more generic and modular data structures, such as edge objects with multiple attributes, which provide flexibility at the cost of increased memory and computational requirements.

  Graph Relabeling In PIGALE, the graph is relabeled after the first DFS run to ensure that edges are assigned labels reflecting the order in which they will be traversed during later phases of the algorithm. This relabeling process ensures a

consistent and efficient handling of edges throughout the algorithm. While PIGALE adopts this relabeling strategy, our implementation retains the original labels from the input graph, simplifying the mapping between input and output. During the embedding phase, PIGALE reverts the relabeling to align the output graph with the original input graph.

Constraint Stack The data structure for the constraint stack is nearly identical in both implementations, comprising conflict pairs that define intervals with high and low edges on either the left or right side. However, the operations performed on the stack differ significantly between the two implementations.

- **Best Edge Placement:** Brandes always merges and aligns edges when possible and not only when required by constraints. Therefore when a new edge does not fit in the stack, no further checks are needed, because all other edges are already in the best spot and the graph can be declared non-planar.
- **Handling of Lowpoint Edges:** A critical difference is the handling of lowpoint edges. Brandes' approach ensures lowpoint edges are placed on the first and not handling constraints for it. Therefore when handling a constraint, the algorithm already knows that a lowpoint edge exists and does not have to check. In PIGALE, lowpoint edges are treated like other back edges, leading to more complex operations that require additional conditions and checks.
- **Generalized vs. Case-Specific Design:** Brandes' algorithm adopts a generalized approach, removing the need for many case-specific operations. In contrast, PIGALE explicitly handles specific cases, such as storing pointers for forks and resolving constraints immediately. This difference reflects their contrasting design philosophies: Brandes prioritizes a unified, theoretical perspective, while PIGALE focuses on a practical, step-by-step methodology.

Final Insights While Brandes' algorithm is easier to read and follow, as it avoids numerous case distinctions, the PIGALE implementation is more straightforward in practice, allowing developers to focus on specific edge cases without needing to consider the entire graph. Additionally, PIGALE's data structures are highly optimized for memory allocation and computational efficiency. On the other hand, Brandes' implementation eliminates redundant steps, streamlining the planarity testing process. Both approaches offer valuable insights into planarity testing, balancing theoretical elegance with practical application.

CHAPTER 6

Empirical Evaluation

This chapter evaluates the performance of our implementation of Brandes' Left-Right planarity testing algorithm. We conduct empirical tests comparing it to other implementations to measure runtime efficiency and examine the impact of Brandes refinements on computational performance.

We describe the testing methodology, the datasets used, and the benchmarking process. By analyzing the results, we determine how well the Left-Right approach performs in practice and how it compares to existing implementations.

6.1 Test data

To evaluate the efficiency of the algorithms, we needed a diverse and structured set of graphs for testing. The focus was primarily on planar graphs since non-planar inputs could introduce inconsistencies in runtime comparisons. Non-planar graphs may cause one algorithm to detect planarity obstructions early while others process most of the graph before reaching a conclusion, leading to unbalanced results. Thus, planar graphs provide a more meaningful basis for comparison.

The test data set consists of five types of graphs: single connected graphs with $n-1$ edges (forming trees), random connected planar graphs with $2n$ edges, random triconnected planar graphs with $3n-6$ edges (maximally dense planar graphs), wheel graphs, and grid graphs. These types were chosen to ensure variability in structure and density, allowing for a thorough analysis of algorithm performance under different conditions. All graphs were generated using the OGDF library in C++, leveraging its robust random graph generation capabilities. Each graph type was represented by 200 files, with sizes ranging from 5000 to 1,000,000 nodes, incremented in steps of 5000. For grid graphs, the size followed an $n \times 1000$ pattern, where n ranged from 5 to 1000, ensuring node counts matched the range of the other graph types.

The graphs were saved in GraphML format for compatibility with the OGDF and were additionally converted into simple text files. These text files contained the number of nodes and edges, followed by a line for each edge specifying its starting and target nodes. This simplified format facilitated easy integration with other testing frameworks and algorithms.

By focusing on planar graphs and including a variety of graph types, the test data set provides a balanced and representative sample for efficiency evaluations. The exclusion of non-planar graphs ensures that runtime comparisons reflect the algorithms' behavior on well-defined problem instances, avoiding inconsistencies that could arise from unpredictable planarity obstructions.

6.2 Testing Set-up

In our testing setup, we compared four different algorithms implemented in C++ libraries: the left-right approach from the Pigale framework (2003) [dFdM02] by De Fraysseix and Rosenstiehl, the implementation by Boyer and Myrvold [BM06] in the OGDF framework, and two versions of our own implementation based on the pseudo-code by Brandes (2009) for the left-right approach. The two versions of our implementation differ in their embedding steps: one with a quadratic runtime and the other with a linear runtime.

All implementations are compiled as standalone executables that take as input the path to a graph file and a boolean flag indicating whether to run with or without embedding. Each implementation executes the planarity test four times for every graph, and the median runtime is used for comparison. The output is a string that contains either '0' or '1' (indicating whether the graph is non-planar or planar, respectively) followed by the runtime for each of the four runs. The time measurement is conducted using the 'chrono' library, capturing only the duration of the planarity test and embedding steps. File reading and logging operations are excluded from the measured time.

To streamline execution, we created five Python scripts, one for each graph type, which call all four executables once with embedding and once without embedding. The scripts record the runtimes in CSV files. Another Python script processes these CSV files, generating individual plots as PNG files for each graph type and a consolidated Markdown file summarizing the results.

The tests were conducted on a SLURM cluster, where batch jobs were executed per graph type. This allowed us to leverage parallel processing across multiple nodes. The cluster features Intel Xeon E5-2690v2 CPUs (10 cores at 3.00 GHz) with 17 nodes, each equipped with 64 GiB of RAM (node 17 was excluded due to differences in hardware). Each job was allocated 1 GiB of RAM and executed on a single core to ensure sequential processing. The cluster runs on Linux kernel version 6.1.0-18-amd64.

6.3 Results

For a more detailed comparison, we evaluated the test results by splitting them into the graph classes and densities defined in Section 6.1. In the following plots, the Boyer-Myrvold implementation in the OGDF framework is labeled as 'BM', our implementations based on Brandes' pseudo-code are labeled as 'Brandes' (linear embedding step) and 'Brandesq' (quadratic embedding step), and the left-right approach from the Pigale framework is labeled as 'Pigale'.

All implementations are included in the plots for both planarity testing and embedding. However, since the 'BM' implementation performs the embedding step as part of the planarity test, it is inherently included in the runtime for 'BM' in both types of plots. This makes 'BM' directly comparable to the other algorithms only in the plots with embedding. In the plots without embedding, 'Brandes' and 'BrandesQ' show nearly identical performance, as the primary difference between the two implementations lies in their embedding steps.

In the plots, the x-axis represents the number of vertices in millions (10^6), and the y-axis shows the runtime in seconds. This provides a clear evaluation of algorithm performance across different graph sizes and densities.

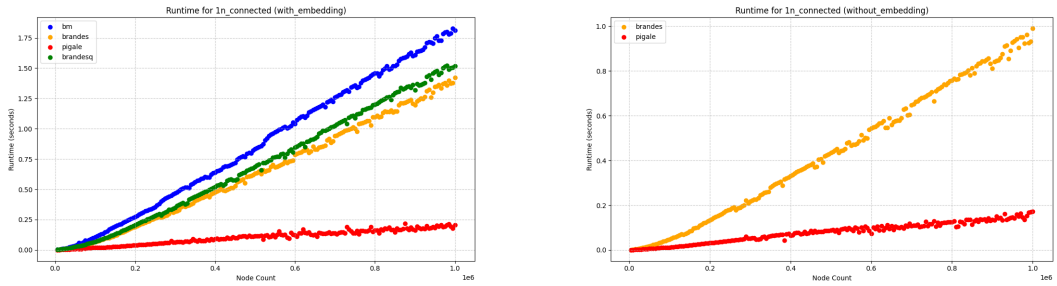
6.3.1 Evaluation of Single Connected Graphs

The evaluation of the runtime for the algorithms on `single connected` graphs, which represent trees ($m = n - 1$), reveals consistent trends (see Figures 6.1, 6.2, and 6.3). `Pigale` is the fastest algorithm across all configurations, achieving runtimes approximately 70% faster than `BM`. `Brandes` performs closer to `BM` but is still about 55% faster. The quadratic embedding step in `BrandesQ` introduces a small efficiency loss compared to `Brandes`, but this loss remains minimal for this graph type.

When considering embedding overhead (see Figure 6.2), `Pigale` achieves the best embedding overhead, with an increase of only around 20%. It is also worth mentioning that the quadratic embedding step for `BrandesQ` is about 15% slower than `Brandes`, even though no quadratic runtime trend is observed. `Brandes` exhibits a consistent runtime increase of around 20–30%, comparable to `Pigale`'s overhead.

The speed-up plots (see Figure 6.3) highlight that while `BM` and `Brandes` show a similar and more consistent trend in their speed-ups with an increasing number of nodes, `Pigale` is more inconsistent across different graph sizes. The best performance of `Pigale` is observed around 100,000 nodes (see Figure 6.3a), but it becomes more consistent for larger graphs.

Overall, `Pigale` maintains the highest efficiency for this graph class, while `Brandes` provides a strong alternative with consistent performance and manageable embedding overhead. `BrandesQ` remains competitive, with only minor efficiency losses due to its quadratic embedding step.



(a) Runtime with embedding. (b) Runtime without embedding.

Figure 6.1: Runtime for `single connected` graphs with (a) embedding and (b) without embedding.

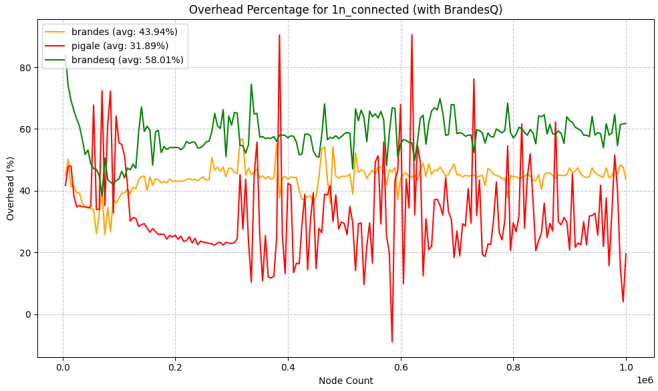
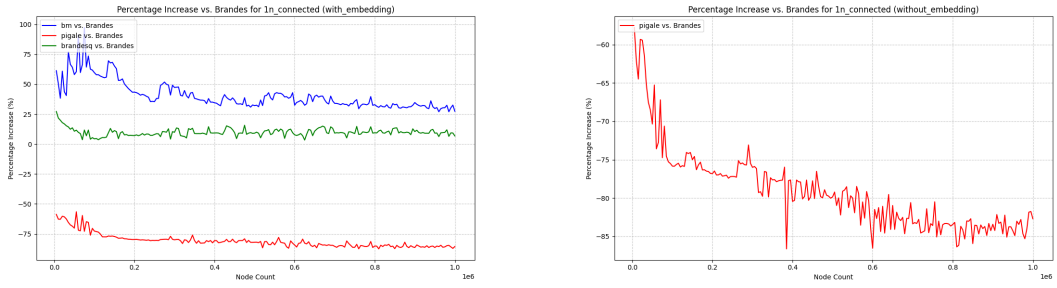


Figure 6.2: Overhead percentage with embedding for `single connected` graphs (including `BrandesQ`).



(a) Speed-up with embedding. (b) Speed-up without embedding.

Figure 6.3: Speed-up vs. `Brandes` for `single connected` graphs with (a) embedding and (b) without embedding.

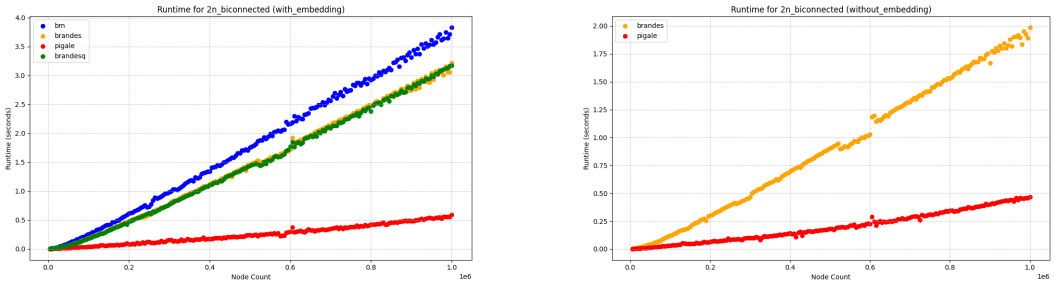
6.3.2 Evaluation of Biconnected Graphs with $2n$ Edges

The evaluation of the runtime for the algorithms on `2n_biconnected` graphs, which are biconnected graphs with $m = 2n$, shows similar results to those observed for `single connected` graphs (see Figures 6.4, 6.5, and 6.6).

The embedding overhead for `BrandesQ` and `Brandes` is nearly identical, indicating that the quadratic embedding step does not cause additional runtime overhead for this graph type. Interestingly, `Pigale` and `BrandesQ` exhibit comparable overhead percentages, while the overhead for `Brandes` decreases slightly, reflecting an improvement in its efficiency. However, the performance of `Brandes` with embedding appears to decline for this graph type, resulting in `Pigale` being approximately 80% faster and `BM` being about 40% slower than `Brandes`.

The speed-up plots (see Figure 6.6) further confirm these trends. `Pigale` continues to demonstrate its efficiency, particularly for larger graphs, while `Brandes` and `BM` maintain a more consistent trend across different graph sizes.

Overall, the results for `2n_biconnected` graphs emphasize the growing efficiency gap between `Pigale` and the other algorithms as graph complexity increases, with `BrandesQ` remaining competitive but showing minimal differences compared to `Brandes`.



(a) Runtime with embedding. (b) Runtime without embedding.

Figure 6.4: Runtime for `2n_biconnected` graphs with (a) embedding and (b) without embedding.

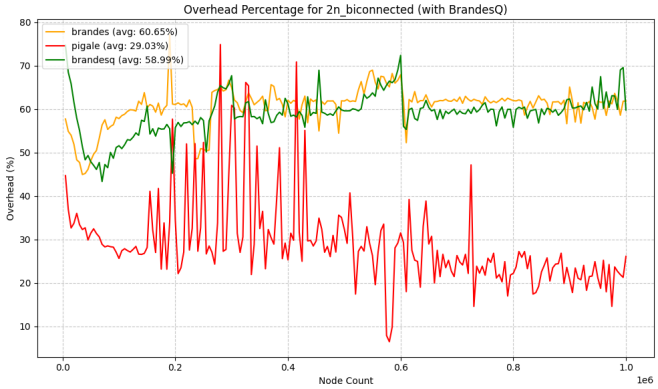
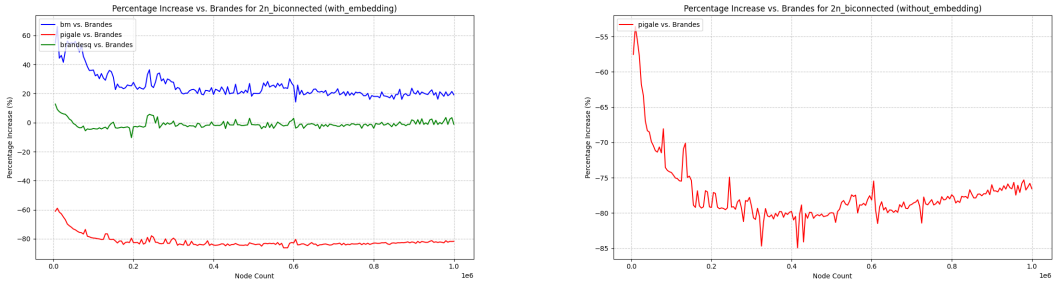


Figure 6.5: Overhead percentage with embedding for `2n_biconnected` graphs (including `BrandesQ`).



(a) Speed-up with embedding.

(b) Speed-up without embedding.

Figure 6.6: Speed-up vs. `Brandes` for `2n_biconnected` graphs with (a) embedding and (b) without embedding.

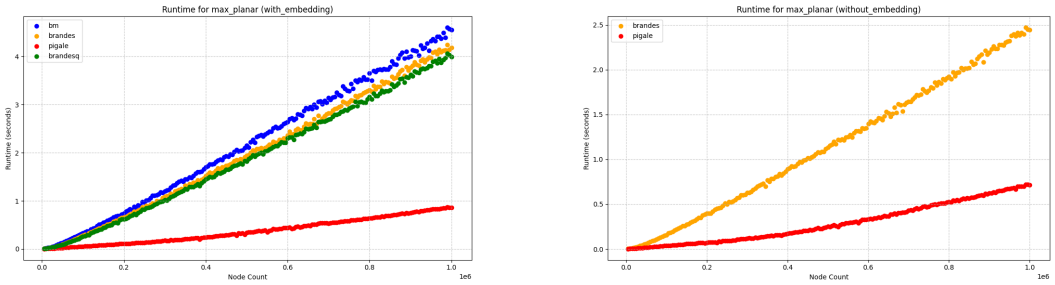
6.3.3 Evaluation of Maximal Planar Graphs with $3n - 6$ Edges

The evaluation of the runtime for the algorithms on `max_planar` graphs, which are maximal planar graphs with $m = 3n - 6$, shows similar results to those observed for `single connected` and `2n_biconnected` graphs (see Figures 6.7, 6.8, and 6.9).

However, the embedding step for `Brandes` further increases in runtime for this graph type, leading to the interesting result that `BrandesQ` becomes faster than `Brandes` with embedding. This trend highlights the worsening performance of the embedding step in `Brandes` as graph structures become more complex. Despite this, `BM` performs relatively better for maximal planar graphs, being only 20% slower than `Brandes`.

The speed-up plots (see Figure 6.9) confirm these observations. `Pigale` remains the fastest algorithm, maintaining its efficiency across larger graphs, while `Brandes` continues to show declining performance in embedding steps for this more complex graph type.

Overall, the results for `max_planar` graphs emphasize the increasing challenges faced by `Brandes` for embedding steps as graph complexity grows, while `Pigale` and `BrandesQ` continue to demonstrate strong performance.



(a) Runtime with embedding. (b) Runtime without embedding.

Figure 6.7: Runtime for `max_planar` graphs with (a) embedding and (b) without embedding.

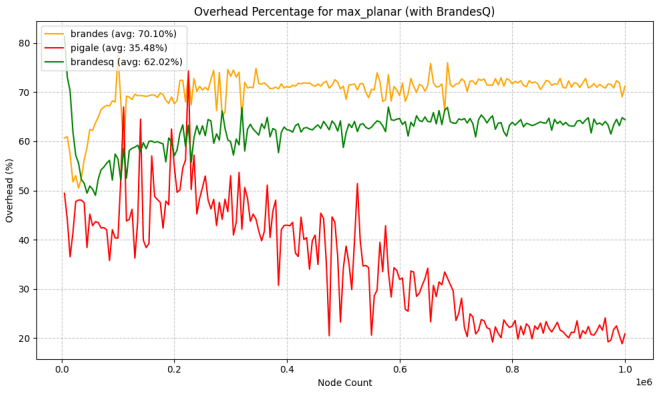
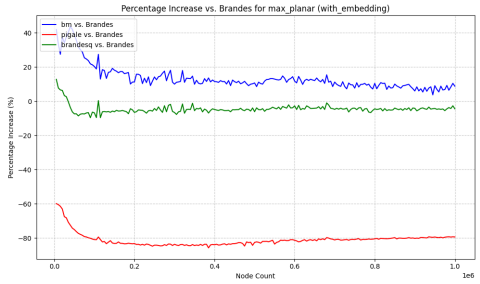
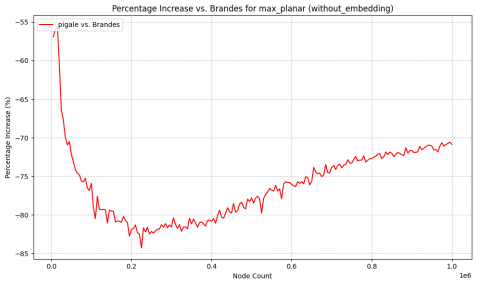


Figure 6.8: Overhead percentage with embedding for `max_planar` graphs (including `BrandesQ`).



(a) Speed-up with embedding.



(b) Speed-up without embedding.

Figure 6.9: Speed-up vs. `Brandes` for `max_planar` graphs with (a) embedding and (b) without embedding.

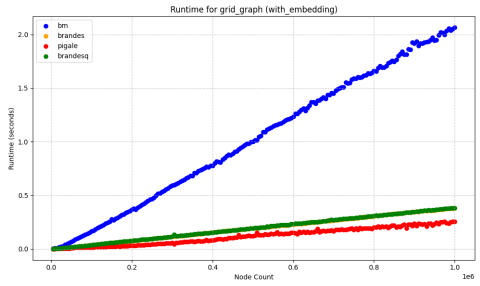
6.3.4 Evaluation of Grid Graphs

The evaluation of the runtime for the algorithms on `grid_graph` graphs reveals notable differences compared to the other graph types (see Figures 6.10, 6.11, and 6.12).

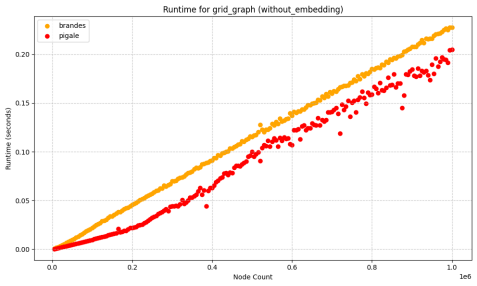
Without embedding, `Brandes` demonstrates excellent performance, almost reaching the efficiency of `Pigale`, especially for larger graphs, where it is only about 10% slower. With embedding, the performance of both `Brandes` and `BrandesQ` remains strong, but their embedding overhead remains high with around 60%, compared to only 20% for `Pigale`.

`BM`, on the other hand, performs extremely poorly for this graph type, being approximately five times slower than both `Brandes` and `Pigale`. The speed-up plots (see Figure 6.12) clearly illustrate this disparity, highlighting the inefficiency of `BM` on grid graphs compared to the other algorithms.

Overall, the results for `grid_graph` graphs showcase the scalability and efficiency of `Brandes` and `BrandesQ`, even with embedding, while emphasizing the poor suitability of `BM` for this structured graph type.



(a) Runtime with embedding.



(b) Runtime without embedding.

Figure 6.10: Runtime for `grid_graph` graphs with (a) embedding and (b) without embedding.

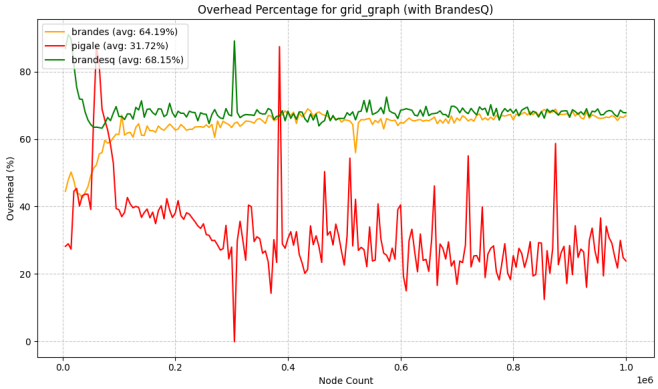
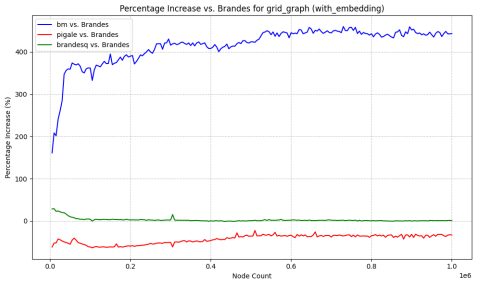
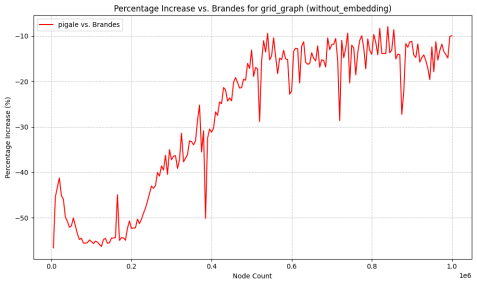


Figure 6.11: Overhead percentage with embedding for `grid_graph` graphs (including `BrandesQ`).



(a) Speed-up with embedding.



(b) Speed-up without embedding.

Figure 6.12: Speed-up vs. `Brandes` for `grid_graph` graphs with (a) embedding and (b) without embedding.

6.3.5 Evaluation of Wheel Graphs

The evaluation of the runtime for the algorithms on `wheel_graph` graphs reveals similar trends to those observed for `grid_graph` graphs (see Figures 6.13, 6.14, and 6.15).

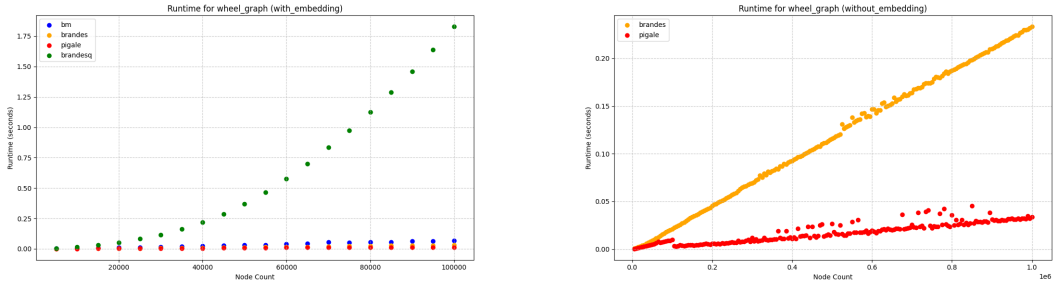
However, due to the quadratic runtime of `BrandesQ`, embedding on wheel graphs was only tested with up to 100,000 nodes. Beyond this threshold, the rapidly increasing overhead would make computations impractical and limit meaningful comparisons, as the performance gap between `BrandesQ` and the other methods would grow excessively large. This restriction ensures that the evaluation remains interpretable while still capturing the significant efficiency differences observed across the tested graph sizes.

`Brandes`, `Pigale`, and `BM` demonstrate results similar to grid graphs, with `Brandes` and `Pigale` being extremely fast on these structures, while `BM` remains relatively slow. However, the embedding step for `BrandesQ` performs exceptionally poorly on wheel graphs, resulting in an embedding overhead factor of up to 50 for the largest tested graph

sizes, with increasing trends. This overhead clearly demonstrates the quadratic runtime of `BrandesQ`, as the overhead increases with the node count.

The speed-up plots (see Figure 6.15) further emphasize the inefficiency of `BrandesQ` for embedding on this graph type. Meanwhile, `Brandes` and `Pigale` continue to perform efficiently, even for large graph sizes, showcasing their suitability for highly structured graph types like wheel graphs.

Overall, the results for `wheel_graph` graphs highlight the scalability and robustness of `Brandes` and `Pigale` for this graph type, while revealing the significant limitations of `BrandesQ` due to its quadratic embedding step.



(a) Runtime with embedding. (b) Runtime without embedding.

Figure 6.13: Runtime for `wheel_graph` graphs with (a) embedding and (b) without embedding.

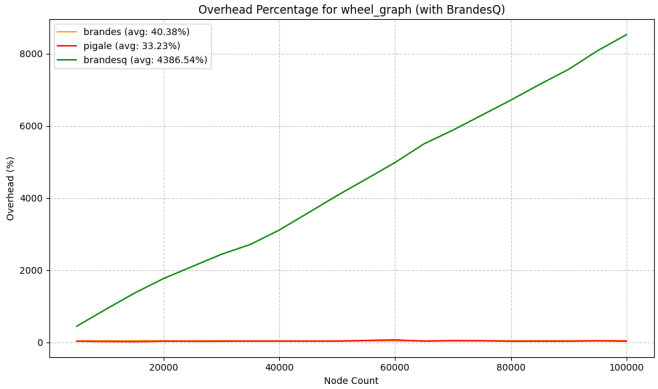
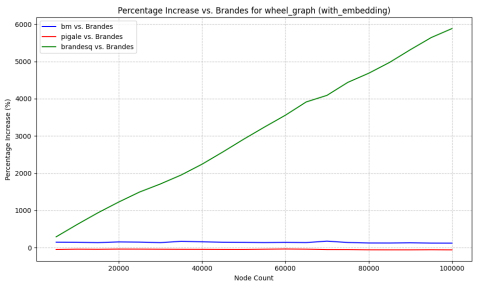
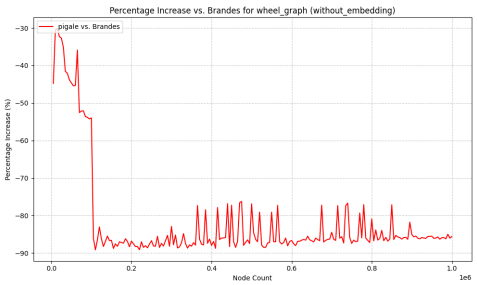


Figure 6.14: Overhead percentage with embedding for `wheel_graph` graphs (including `BrandesQ`).



(a) Speed-up with embedding.



(b) Speed-up without embedding.

Figure 6.15: Speed-up vs. `Brandes` for `wheel_graph` graphs with (a) embedding and (b) without embedding.

6.3.6 Summary of Efficiency Tests

This chapter evaluated the performance of various planarity testing and embedding algorithms across different graph types. The results highlight significant differences in efficiency, particularly when considering runtime, embedding overhead, and graph structure. The following key insights summarize the findings:

- **Pigale as the Fastest Implementation:** The Pigale implementation consistently achieved the best runtime across all tested graph types. Its efficient design and low embedding overhead, averaging around 20%, make it the most versatile and high-performing choice for planarity testing and embedding.
- **Strong Performance of Brandes:** The Brandes implementation performed relatively well, especially for structured graph types such as wheel graphs and grid graphs. In these cases, Brandes closely approaches the efficiency of Pigale, being only about 10% slower in configurations without embedding.
- **Limitations of Brandes in Embedding Steps:** While Brandes excelled in runtime performance, its embedding step remains a notable limitation. On average, the embedding step incurred an overhead of 60%, significantly higher than the 20% overhead observed in Pigale. This drawback affects its overall efficiency, particularly for scenarios requiring frequent embedding.
- **BM as the Slowest Implementation:** The Boyer-Myrvold (BM) implementation consistently demonstrated the slowest runtime among all tested algorithms. Unlike Brandes, BM showed no significant performance improvements for structured graph types such as wheel graphs or grid graphs. However, for more complex graph types, such as maximal planar graphs, BM's performance was comparable to that of Brandes.
- **Quadratic Runtime of BrandesQ:** The BrandesQ implementation provided embedding performance comparable to Brandes and demonstrated more consistent

behavior for complex graph types. However, its quadratic runtime in the embedding step severely limits its applicability. This inefficiency becomes particularly apparent for random graphs where edges are unevenly distributed among nodes, as well as for structured graph types like wheel graphs, where the embedding overhead increases significantly with graph size.

In conclusion, Pigale emerges as the most efficient algorithm for planarity testing and embedding across a wide range of graph types. Brandes provides a competitive alternative for structured graphs, but its embedding overhead is a critical limitation. BM, while consistently slower, remains a viable option for complex graph types. Finally, BrandesQ, despite its consistency, is limited by its quadratic embedding step, making it unsuitable for certain graph configurations.

CHAPTER 7

Discussion

This chapter analyzes the key findings of this thesis, highlighting the efficiency of Brandes refinements to the Left-Right planarity testing approach. It compares implementations, explores broader implications, and outlines future research directions.

7.1 Summary and Interpretation of Key Findings

This thesis provides an in-depth examination of the Left-Right planarity testing approach, focusing on its evolution, implementation, and efficiency. Our implementation of Brandes version of the algorithm demonstrates superior performance compared to other state-of-the-art implementations when executed within the same computational framework. This suggests that the refined algorithm introduces optimizations that enhance efficiency while maintaining practical feasability.

A key observation is that while the PIGALE implementation benefits from an optimized data structure and framework efficiency, it still outperforms our implementation overall. However, despite PIGALEs structural advantages and lower overhead, our implementation comes close in performance, showing that Brandes' refinements allow for a competitive execution even without the highly optimized framework of PIGALE. This highlights the distinction between framework-dependent optimizations and algorithm-intrinsic improvements.

One of the most notable algorithmic differences in Brandes' refinement is how the traversal order allows the algorithm to bypass the validation of certain conditions, as they are inherently satisfied due to the structured processing sequence. This reduces unnecessary checks and contributes to a more efficient execution. Additionally, Brandes approach is more generalized and avoids excessive case distinctions, making it not only more streamlined but also easier to read and understand.

Finally, the structured processing order in Brandes approach may have broader implications beyond planarity testing. Similar techniques could be applied to other graph traversal problems where case minimization and order enforcement improve efficiency and readability. Exploring these connections in future research could provide additional insights into the broader applicability of these refinements.

7.2 Future Research Directions

While this thesis has provided valuable insights into the efficiency and structure of Brandes' refinements of the Left-Right planarity testing approach, several areas remain open for further exploration.

One key avenue for future research is the integration of Brandes algorithm within the PIGALE framework, utilizing its optimized data structures. This would allow for a direct performance comparison with the current PIGALE implementation, helping to determine whether Brandes' refinements can outperform the existing implementations in this framework.

Additionally, implementing Brandes' algorithm in other high-performance frameworks beyond OGDF would provide further insights into its efficiency. By benchmarking against other planarity testing algorithms across different frameworks, it would be possible to assess whether Brandes approach can surpass current implementations in different computational environments.

Another important extension would be the implementation of Kuratowski subgraph extraction for non-planar graphs. While this thesis focused solely on planarity testing, practical applications often require identifying the exact obstruction preventing a graph from being planar. Adding this functionality would increase the practical relevance of the Left-Right approach for real-world use cases.

Finally, a deeper examination of why the Left-Right approach has not seen widespread adoption in real-world applications, despite its strong performance, would be valuable. Investigating industry use cases, developer preferences, and potential barriers to implementation could provide insights into whether the method requires further refinements or simply better integration into existing software tools.

By addressing these questions, future research can further clarify the strengths and limitations of Brandes' refinements and explore new opportunities for improving and applying the Left-Right planarity testing approach.

Overview of Generative AI Tools Used

In the course of this work, various AI tools were utilized to support different aspects of the research and implementation. These tools assisted in analyzing, structuring, and refining content while ensuring accuracy.

Notebook LM

- Used for additional analysis in the comparison of Brandes' work and Rosenstiehl and de Fraysseixs work.

ChatGPT

ChatGPT was employed for a variety of tasks, ranging from content formulation to implementation support:

Textual Refinements

- Rephrasing and improving sentence structures
- Expanding and articulating content
- Structuring the introduction based on the thesis outline
- Generating summaries from pre-selected key points
- Creating the abstract

Technical Assistance

- Debugging support for the implementation
- Code refactoring to improve clarity

- Generating Python run scripts for efficiency tests
- Creating LaTeX graphs
- Refining plain text into LaTeX format

Definition Generation

ChatGPT was also used to provide precise definitions for fundamental graph theory concepts, including:

- Depth-First Search (DFS)
- Euler's Formula
- Kuratowskis Theorem
- Basic graph fundamentals

All AI-generated texts and outputs were carefully reviewed, refined through targeted prompts, or manually revised to ensure accuracy.

List of Figures

2.1 Illustration of Connected Components, Biconnectivity, and a Cut Vertex. 6
2.2 Simple undirected planar graph 9
2.3 DFS-tree of the graph, with tree edges in black and back edges in red . . 9
2.4 Complete graph K_5 9
2.5 Subdivision of $K_{3,3}$ 9
2.6 Example of return edges. Note that b_2 and b_3 are return edges for e_4, but only b_3 is a return edge for e_2. 10
2.7 Example graph showing same-constraints in the same color and different-constraints in different colors. 12

3.1 Example graph K_5 in a state where the merging operation halts due to non-planarity. 20
3.2 Example graph in a state where conflict handling halts due to non-planarity. 22
3.3 Graph showing the order of outgoing edges around vertex v. 23

4.1 Example graph (from Section 2.4) illustrating key terms. See details below. 29

6.1 Runtime for `single connected` graphs with (a) embedding and (b) without embedding. 50
6.2 Overhead percentage with embedding for `single connected` graphs (including `BrandesQ`). 50
6.3 Speed-up vs. `Brandes` for `single connected` graphs with (a) embedding and (b) without embedding. 50
6.4 Runtime for `2n_biconnected` graphs with (a) embedding and (b) without embedding. 51
6.5 Overhead percentage with embedding for `2n_biconnected` graphs (including `BrandesQ`). 52
6.6 Speed-up vs. `Brandes` for `2n_biconnected` graphs with (a) embedding and (b) without embedding. 52
6.7 Runtime for `max_planar` graphs with (a) embedding and (b) without embedding. 53
6.8 Overhead percentage with embedding for `max_planar` graphs (including `BrandesQ`). 53

6.9 Speed-up vs. `Brandes` for `max_planar` graphs with (a) embedding and (b) without embedding. 54
6.10 Runtime for `grid_graph` graphs with (a) embedding and (b) without embedding. 54
6.11 Overhead percentage with embedding for `grid_graph` graphs (including `BrandesQ`). 55
6.12 Speed-up vs. `Brandes` for `grid_graph` graphs with (a) embedding and (b) without embedding. 55
6.13 Runtime for `wheel_graph` graphs with (a) embedding and (b) without embedding. 56
6.14 Overhead percentage with embedding for `wheel_graph` graphs (including `BrandesQ`). 56
6.15 Speed-up vs. `Brandes` for `wheel_graph` graphs with (a) embedding and (b) without embedding. 57

Bibliography

[AP61] Louis Auslander and S.V. Parter. On imbedding graphs in the sphere. *Journal of Mathematics and Mechanics*, pages 517–523, 1961.

[BCPDB03] John M Boyer, Pier Francesco Cortese, Maurizio Patrignani, and Giuseppe Di Battista. Stop minding your ps and qs: Implementing a fast and simple dfs-based planarity testing and embedding algorithm. In *Proceeding of the Xth International Symposium on Graph Drawing*, pages 25–36. Springer, 2003.

[BL76] Kellogg S Booth and George S Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of computer and system sciences*, 13(3):335–379, 1976.

[BM99] John M Boyer and Wendy J Myrvold. Stop minding your p's and q's: A simplified o(n) planar embedding algorithm. In *Proceeding of the SODA*, pages 140–146, 1999.

[BM06] John M Boyer and Wendy J Myrvold. Simplified o(n) planarity by edge addition. *Graph Algorithms Appl*, 5:241, 2006.

[Bra09] Ulrik Brandes. The left-right planarity test. *Manuscript submitted for publication*, 2009.

[CGJ+13] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). *Handbook of graph drawing and visualization*, 2011:543–569, 2013.

[dF08] Hubert de Fraysseix. Trémaux trees and planarity. *Electronic Notes in Discrete Mathematics*, 31:169–180, 2008.

[dFdM02] Hubert de Fraysseix and Patrice Ossona de Mendez. Pigale-public implementation of a graph algorithm library and editor. *SourceForge project page http://sourceforge. net/projects/pigale*, 2002.

[dFdM12] Hubert de Fraysseix and Patrice Ossona de Mendez. Trémaux trees and planarity. *European Journal of Combinatorics*, 33(3):279–293, 2012.

[dFdMR06] Hubert de Fraysseix, Patrice Ossona de Mendez, and Pierre Rosenstiehl. Trémaux trees and planarity. *International Journal of Foundations of Computer Science*, 17(05):1017–1029, 2006.

[dFR82] Hubert de Fraysseix and Pierre Rosenstiehl. A depth-first-search characterization of planarity. In *North-Holland Mathematics Studies*, volume 62, pages 75–80. Elsevier, 1982.

[dFR85] Hubert de Fraysseix and Pierre Rosenstiehl. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5:127–135, 1985.

[Eul22] Leonhard Euler. *Opera omnia*, volume 14. Typis et in aedibus BG Teubneri, 1922.

[HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[Kur30] Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fundamenta mathematicae*, 15(1):271–283, 1930.

[Lem67] Abraham Lempel. An algorithm for planarity testing of graphs. In *Proceeding of the Theory of Graphs: International Symposium.*, pages 215–232. Gorden and Breach, 1967.

[MM96] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[Pat13] Maurizio Patrignani. Planarity testing and embedding., 2013.

[Tam13] Roberto Tamassia. Graph drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 19, pages 469–490. CRC Press, 2013. Section discussing PIGALE.