Evaluation and Comparison of Planarity Algorithms

Bachelor Thesis of

Julian Huber

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

UNIVERSITÄT PASSAU

Reviewer:	Prof. Dr. Ignaz Rutter
Advisor:	Matthias Pfretzschner, M. Sc.

Time Period: 21st December 2023 – 21st March 2024

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Passau, August 2, 2024

Abstract

Planarity testing is one of the most important problems in graph theory. There are several linear-time algorithms available that solve this problem, each of them using different methods. This thesis describes the algorithms by Booth and Lueker [BL76], Boyer and Myrvold [BM04], and Haupler and Tarjan [HT08] with a focus on the Boyer and Myrvold algorithm (since we provide a port to the OGDF library [CGJ$^+$13] for it, to improve the efficiency of an already existing implementation, as ours is closer to the reference implementation). We then compare the performance in an extensive test. We observe that newer algorithms in general perform better as the algorithm by Haeupler and Tarjan is the fastest OGDF-based implementation in every tested scenario. Except for grid graphs, our port of the Boyer and Myrvold algorithm is the second-fastest OGDF-based implementation, outperforming the existing one. The implementation of the Booth and Lueker algorithm is in general the slowest. However, it is faster than the Boyer and Myrvold implementations for grid graphs.

Deutsche Zusammenfassung

Das Testen auf Planarität ist eines der wichtigsten Probleme der Graphentheorie. Es gibt einige Linearzeitalgorithmen, die das Planaritätsproblem für Graphen unter Verwendung leicht unterschiedlicher Methoden lösen. Diese Arbeit beschreibt die Funktionsweise der Planaritätsalgorithmen von Booth und Lueker [BL76], Boyer und Myrvold [BM04] und Haeupler und Tarjan [HT08], wobei der Fokus auf dem Algorithmus von Boyer und Myrvold liegt (da wir dafür eine Portierung zur OGDF-Bibliothek [CGJ$^+$13] bereitstellen, um die Effizienz einer bereits existierenden Implementierung zu verbessern, da unsere Portierung näher an der Referenzimplementierung ist). Wir vergleichen die Performanz der Implementierungen der beschriebenen Algorithmen, wobei sich zeigt, dass Implementierungen neuerer Algorithmen in der Regel am schnellsten sind. So ist die Implementierung des Algorithmus von Haeupler und Tarjan in allen getesteten Szenarien die schnellste OGDF-basierte Implementierung, gefolgt von unserer Portierung des Boyer-Myrvold Algorithmus. Die Implementierung des Booth-Lueker Algorithmus ist im Allgemeinen am Langsamsten, bei Gittergraphen hingegen aber schneller als unsere Implementierung des Boyer-Myrvold Algorithmus.

Contents

1. Introduction

Let $G = (V, E)$ be a simple graph. G is considered to be *planar*, if there exists a drawing Γ of G so that no two edges in E intersect, except in their common endpoints. The problem that arises from that is to find out if such a drawing exists for a give graph, which is one of the most important problems in graph theory.
The knowledge of a graph being planar is quite helpful for a variety of other graph problems, for example, chip design or rail track planning. Beyond that, it is shown that any planar graph is 4-colorable [AH77, AHK77, RSST97], which is applicable when coloring maps. Furthermore, for general graphs, the 3-Coloring problem has been proven to be $\mathcal{NP}$-complete but if we have a planar graph without triangles, Grötzsch [Gro59] showed that a 3-coloring always exists, while Dvorak, Kawarabayashi and Thomas introduced a method how this can be found in linear time [DKT09]. Another graph problem that is simplified through planarity is the Clique-problem, which is known to be $\mathcal{NP}$-complete in general, but takes polynomial time for a planar graph [Pat13].
Thus, methods for testing a graph's planarity have been explored and revised for many years now, starting from polynomial-time algorithms (for example Aulander and Parter [AP61]), followed by the first linear time planarity algorithm introduced by Hopcroft and Tarjan in 1974 [HT74]. Further improvements and simplifications using different approaches proceeded, intending to make planarity testing as efficient as possible.

1.1 Related Work

Patrignani [Pat13] provided an overview of the topic of planarity testing by introducing theoretic fundamentals and describing the known techniques for planarity testing and the most important planarity algorithms. The first planarity algorithms were published by Auslander and Parter [AP61], Goldstein [Gol63], Bader [Bad64], and Even, Lempel, and Cederbaum [LEC67], which, however, had a superlinear time complexity. As already mentioned, Hopcroft and Tarjan [HT74] introduced the first linear time planarity algorithm which was more a theoretical construct due to its complexity. Booth and Lueker [BL76] two years later published their much simpler algorithm, using PQ-trees. Shih and Hsu [SH99], Boyer and Myrvold [BM99, BM04], and Haeupler and Tarjan [HT08] provided the newest approaches to planarity testing, following the idea by Booth and Lueker.
As to go beyond simple planarity, Ringel [Rin65] first described the problem of *1-Planarity* which asks whether a graph can be drawn in the plane so that each edge has at most one crossing. While simple planarity can be done in linear time, 1-Planarity has been proven

to be $\mathcal{NP}$-hard [KM08]. Furthermore, for *Simultaneous Planarity* [BKR15], two graphs with the same vertex set must have planar drawings where the vertices in each drawing are mapped to the same point in the plane of the drawings. Other interesting related problems are *Clustered Planarity* or *Simultaneous Embedding With Fixed Edges* [FCE95], which is an extension of Simultaneous Planarity [BKR15].

1.2 Contribution and Outline

In this thesis, we explain the basics of planarity and describe some planarity algorithms but also do an extensive evaluation and comparison of the algorithms we described.

In Chapter 3, we begin with an overview of the most important planarity algorithms currently available. In the OGDF library[1], two planarity tests are already implemented, namely the algorithms by Booth and Lueker [BL76], and Boyer and Myrvold [BM04]. Furthermore, the algorithm by Haeupler and Tarjan [HT08] was implemented by the chair of Theoretical Computer science at the University of Passau using the OGDF library. Since these are the algorithms for which there is an OGDF implementation available, we include these in our comparison. Additionally, we ported the reference implementation of the Boyer and Myrvold algorithm that was included in their paper to the OGDF data structure as the existing OGDF implementation of the algorithm has some efficiency issues. We provide a more detailed description of those three algorithms with a strong focus on the algorithm by Boyer and Myrvold.
In Chapter 4, we present a testing framework that we use to make an evaluation and comparison of the OGDF implementations [CGJ+13]. We measure their performance on a large collection of different graphs to obtain an impression of which implementation works best in general, respectively if there are situations, where a specific algorithm is more efficient than the others. We also analyze the slowdown by additionally computing a planar embedding.

[1] `https://ogdf.uos.de/`

2. Preliminaries

To be able to fully understand the theoretical content of this thesis, some fundamental graph concepts and data structures are required. We followed the introduction by Patrignani [Pat13] to give an overview of what we consider essential.

2.1 Graph Theory Fundamentals

A graph G is a pair (V, E), where V is the set of vertices and $E = \{(u, v) | u, v \in V \land u \neq v\}$ are the edges that connect the vertices of G. We call G *undirected* if for any edge $(u, v) \in E$, also $(v, u) \in E$ holds. A sequence of vertices $v_1, v_2, ..., v_k, k \in \mathbb{N}$ where the edges $(v_i, v_{i+1}), i = 1, ..., k - 1$ exist, is called a *path* (from v_1 to v_k). An undirected graph is called *connected* if, for any two vertices u and v, we can find a path that leads from u to v. A *cut vertex* of a connected graph is a vertex whose removal would lead to disconnecting the graph. A graph is *biconnected* if it has no cut vertices, and is *triconnected* if removing any of its vertices leaves a biconnected graph. For two graphs G_1 and G_2, G_2 is a *subgraph* of G_1, if $G_1 \cup G_2 = G_1$ holds. If we consider subsets $V' \subseteq V$ and $E' \subseteq E$ with $E' = \{(u, v) \in E \mid u, v \in V'\}$, the resulting graph $G' = (V', E')$ is called a *subgraph* of G *induced* by V'. The maximal biconnected subgraphs of a (connected) graph are called its *biconnected components* (see Figure 2.1).

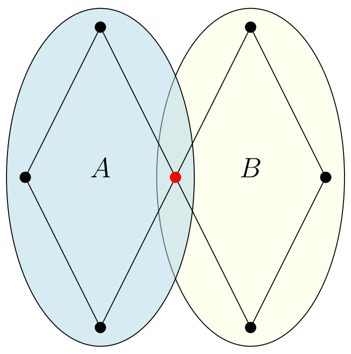


Figure 2.1: A connected graph with the two biconnected components A and B and one cut vertex, which is colored red here.

If we later apply a depth-first-search to a graph and consider its biconnected components, the vertex with the lowest depth-first-index of each biconnected component is called the *root* of the component it belongs to. If we take an edge (u, v) and replace it with a new vertex w and the two edges (u, w) and (w, v), we call this a *subdivision* of (u, v). Thus, if we have two graphs G_1 and G_2 and G_2 is the result of the subdivision of one or more edges of G_1, then G_2 is called a *subdivision* of G_1 (see Figure 2.2).

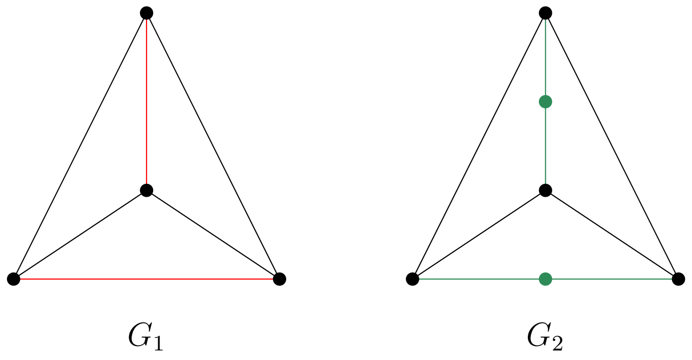


Figure 2.2: G_2 is a subdivision of G_1 since the two green edges in G_1 were subdivided by the green vertices and their corresponding edges in G_2.

2.2 Planar graphs, their properties and characterization

The planarity of a graph is one of the most important problems in graph theory. We call a graph *planar* if it can be drawn in a two-dimensional plane in a way that no two edges cross each other except maybe in their endpoints (if two edges have a common endpoint). A *drawing* of a graph induces an ordering of the edges incident to each vertex of the graph. Two drawings of a graph can be considered *equivalent* if they induce the same ordering of the edges. An equivalence class of drawings is called an *embedding*. If we consider a planar drawing of a graph, we can see that the graph can be divided into regions that are enclosed by a collection of vertices and edges. We call these regions *faces*. The face that encloses the (infinite) area outside the graph is called the *external face* of the graph.

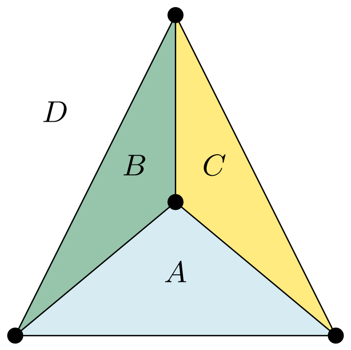


Figure 2.3: This drawing of the complete graph with four vertices has four faces, labeled with A, B, C, and D, where D denotes the external face.

Furthermore, if we have a disconnected graph, one can easily observe that the graph is planar if and only if each of its connected components is planar. This can be extended to a graph's biconnected components.
Planar graphs come with some helpful properties. *Euler's Theorem* states that for a planar graph with n vertices, m edges, and f faces, $n - m + f = 2$ holds. We can derive another formula from this theorem, which tells that the number of edges in a planar graph with n vertices is limited by $3n - 6$. Thus, planar graphs can not be arbitrarily dense. This

also allows us to consider an algorithm to have a linear runtime if its time complexity is $O(n+m)$, since

$$O(n+m) = O(n+3n-6) = O(4n-6) = O(n)$$

holds.

One of the most important characterizations of planar graphs was introduced by Kuratowski [Kur30]. He showed that any non-planar graph contains at least one subdivision of K_5 or $K_{3,3}$, where K_5 is the complete graph with 5 vertices and $K_{3,3}$ is the complete bipartite graph, where each set consists of three vertices (see Figure 2.4). Thus, these are the two smallest non-planar graphs existing.

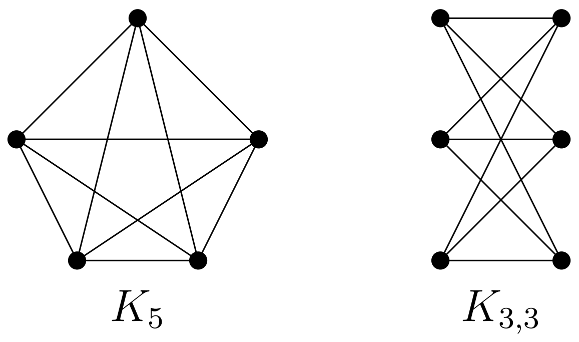


Figure 2.4: The K_5 and the $K_{3,3}$, the basic building blocks of a non-planar graph.

2.3 s,t-numbering for Graphs

An *s, t-numbering* is an important concept by Lempel, Even, and Cederbaum [LEC67] and is needed for some planarity algorithms. It assigns a unique number from 1 to n to each vertex, so that all of them (except vertex 1 and n) are adjacent to at least one vertex with a lower number and at least one vertex with a higher number. Furthermore, vertex 1 and n must also be connected. It was proven that such a numbering exists for every biconnected graph (see example in Figure 2.5). Even and Tarjan [ET76] showed how an s,t-numbering can be computed in linear time.

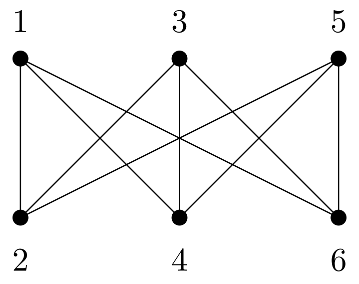


Figure 2.5: A valid s,t-numbering for the $K_{3,3}$.

2.4 The PQ-tree Data Structure

An important data structure, which is essential to some planarity algorithms, is the *PQ-tree*, introduced by Booth and Lueker [BL76]. With a PQ-tree, unstrained permutations of a set of elements can be represented. It is a rooted, ordered tree that consists of *P-nodes*, *Q-nodes*, and *leaf nodes*, where the leaf nodes are the elements of the set over which the

PQ-tree is defined. P and Q-nodes can only appear as non-leaf nodes, whereas the elements of the set can only be leaf nodes. P-nodes allow any possible permutation of its direct children while at a Q-node, we can only reverse their order (see Figure 2.6).

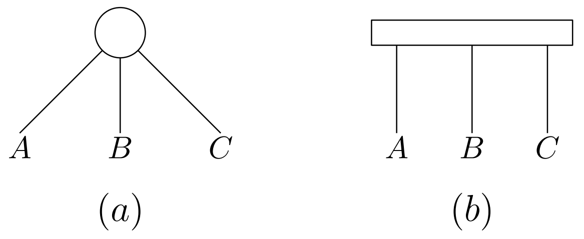


Figure 2.6: We have $U = \{A, B, C\}$. (a) A PQ-tree with a P-node as its root and the elements of U as its children. The possible permutations are $\{(A, B, C), (A, C, B), (B, A, C), (B, C, A), (C, A, B), (C, B, A)\}$. (b) The same PQ-tree as in (a) but with a Q-node as its root instead of a P-node. This yields $\{(A, B, C), (C, B, A)\}$ as permitted permutations.

A valid PQ-tree can also be just a single element, where this leaf node is the only node of the tree, thus its root. Furthermore, if we have a collection of valid PQ-trees where we attach each of their root nodes to a new P-node, we obtain a valid PQ-tree again. The same rule holds if we attach the roots to a new Q-node instead of a P-node. A PQ-tree is called *proper* if each element of the underlying set is included in the tree exactly once, if each P-node has at least two children (since otherwise, we could introduce an arbitrary number of P-nodes as a chain without actually changing the PQ-tree), and if each Q-node has at least three children. Technically, two children would be valid, but it would not make any difference if it was a Q or a P-node. Therefore, this redundancy is removed. If we have a set U, $T(U, U)$ denotes the PQ-tree with one P-node that has all the elements of U as its children.

The only operation that is performed on a PQ-tree is called a *reduction*. Given a subset $S \subseteq U$, a reduction of the original tree T, denoted with $\mathsf{REDUCE}(T,S)$, constructs a new PQ-tree, which contains the additional restriction that the elements of S appear consecutively in the permitted permutations (see Figure 2.7). If it is not possible to make S consecutive with the current tree, the reduction yields the empty PQ-tree. Considering S, a node is called *full*, if every element of S is a leaf of the subtree of the node. The *pertinent subtree* under S is the subtree with minimal height whose leaves contain all the elements of S. We denote its *root* by $\mathsf{ROOT}(T, S)$.

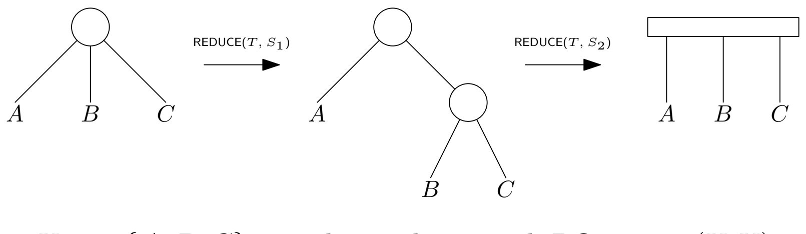


Figure 2.7: On $U = \{A, B, C\}$, we have the initial PQ-tree $T(U, U)$, which is the PQ-tree from Figure 2.6 (a). Let $S_1 = \{B, C\}$ and $S_2 = \{A, B\}$. When we perform $\mathsf{REDUCE}(T, S_1)$, the possible permutations are limited to $\{(A, B, C), (A, C, B), (B, C, A), (C, B, A)$, thus we need to add an extra P-node that makes B and C consecutive. If we then also perform $\mathsf{REDUCE}(T, S_2)$, there are only two possible permutations left, (A, B, C) and (C, B, A), so we introduce a single Q-node as the PQ-tree's root where we attach all the leaves since we only can reverse the pre-order ordering of the tree leaves.

3. Presentation of the Planarity Algorithms

Planarity testing is one of the central problems in theoretical computer science, so there are many approaches to solve this problem. Auslander and Parter [AP61], Goldstein [Gol63], Bader [Bad64], and Lempel, Even, and Cederbaum [LEC67] introduced algorithms with polynomial time complexity. The first $O(n)$ algorithm was described by Hopcroft and Tarjan in 1974 [HT74], which comes with a high level of complexity and is therefore not easy to implement. Other algorithms were proposed subsequently to give a simpler algorithm. Booth and Lueker improved the algorithm of Even, Lempel, and Cederbaum to a liner runtime. More recent approaches include the algorithms by Shih and Hsu [SH99], Boyer and Myrvold [BM99, BM04], and Haeupler and Tarjan [HT08] (for an overview, see Table 3.1). For the comparison, we limited the algorithms to the ones with linear time complexity since they are used in practice, furthermore, a comparison between polynomial and linear algorithms does not provide interesting results.

Algorithm	Year	Time complexity
Auslander & Parter	1961	Polynomial
Goldstein	1963	Polynomial
Bader	1964	Polynomial
Even, Lempel & Cederbaum	1967	$O(n^2)$
Hopcroft & Tarjan	1974	$O(n)$
Booth & Lueker	1976	$O(n)$
Shih & Hsu	1993, 1999	$O(n)$
Boyer & Myrvold	1999, 2004	$O(n)$
Haeupler & Tarjan	2008	$O(n)$

Table 3.1: An overview of different planarity algorithms [Pat13, HT08]

To stay within the scope of this thesis, we chose a small subset of the available planarity algorithms, namely the algorithms by Boyer and Myrvold and Haeupler and Tarjan to have some state-of-the-art algorithms, but we also added the algorithm by Booth and Lueker as one of the older algorithms to see how it performs against those newer ones. This choice was made because we use the OGDF library [CGJ$^+$13][1] as our graph framework, where

[1] `https://ogdf.uos.de/`

BoothLueker and *BoyerMyrvold* are already implemented. Moreover, *HaeuplerTarjan* was implemented by the chair of Theoretical Computer science at the University of Passau. Furthermore, we ported the reference implementation of *BoyerMyrvold* again to the OGDF to potentially eliminate some inefficiency issues of the already existing OGDF implementation, so this implementation is also included in the comparison.

3.1 The algorithm by Booth and Lueker

The BoothLueker algorithm was first introduced by Kellogg S. Booth and George S. Lueker in 1976 [BL76]. They proposed a data structure called a PQ-Tree which is used to represent the permutations of a set U in which various subsets of U occur consecutively. This improves an already existing planarity test by Lempel, Even, and Cederbaum [LEC67] by reducing PQ-Trees instead of using formula manipulation which is a very similar process.

For the algorithm to work, we need an s, t-numbering for the input graph. A valid s, t-numbering is not guaranteed on a connected graph, therefore we begin to split the graph into its biconnected components, which can be done in linear time [AH74]. For those, a valid s, t-numbering exists, as mentioned in Section 2.3. Then it suffices to check, whether each of the biconnected components is planar [BL76], thus the actual algorithm is then executed on every biconnected component. Therefore, without loss of generality, in the following, we assume that the input graph is biconnected. When processing the input, every edge is seen as directed that leads from the lower-numbered vertex to the higher-numbered vertex (according to the s, t-numbering).

In the first step, the algorithm constructs an initial PQ-tree with a P-node as its root and the outgoing edges of vertex 1 as the root's children (lines $1 - 2$ in Algorithm 3.1). Then, the other vertices are processed in ascending order. When processing vertex v, the PQ-tree is reduced to fulfill the constraint that the ingoing edges of v can only appear in a consecutive order. If this is not possible, the result of the reduction operation is an empty PQ-tree, and the input graph is not planar (see lines 4-8 in Algorithm 3.1. Otherwise, we can go on by replacing the pertinent subtree induced by the ingoing edges of v and, if the root is a P-node, all of its descendants with a new PQ-tree that has a P-node as its root and the outgoing edges of v. When this root is a Q-node, we only replace its full children with the new subtree (see lines 11-14 in Algorithm 3.1). After this, we continue with the next vertex. If every vertex (except the last one which does not need to be processed as its incoming edges are already consecutive at this point) has been processed and we still have a proper, non-empty PQ-tree, the input graph is planar.

3.1.1 An Example for the BoothLueker Algorithm

For better understanding, we provide an example of how the algorithm detects non-planarity for $K_{3,3}$. Since $K_{3,3}$ is biconnected, we do not have to split it up any further and can compute an s, t-numbering (see Figure 3.1).

The initial step of the algorithm is to construct a PQ-tree from all the outgoing edges of vertex 1, which in this case are $(1, 2)$, $(1, 4)$, and $(1, 5)$ (see Figure 3.2).

Then we enter the loop. The next vertex that is processed is vertex 2. It has $(1, 2)$ as its only incoming edge, so the tree from Figure 3.2 is already reduced. Thus we simply replace $(1, 2)$ with a new PQ-tree which has the outgoing edges of vertex 2 as its leaves (see Figure 3.3).

Algorithm 3.1: BoothLuekerPlanar

```
   Input: (Biconnected) Graph $G$
   Output: Boolean representing whether $G$ is planar or not
1  Compute $s,t$-numbering for $G$
2  $U \leftarrow$ the set of edges whose lower-numbered vertex is 1
3  $T \leftarrow T(U,U)$
4  for $j \leftarrow 2$ to $n-1$ do
5      $S \leftarrow$ the set of edges whose higher-numbered vertex is j
6      $T \leftarrow$ BUBBLE$(T,S)$
7      $T \leftarrow$ REDUCE$(T,S)$
8      if $T = T(\varnothing,\varnothing)$ then
9          return false
10
11     $S' \leftarrow$ the set of edges whose lower-numbered vertex is j
12     if $ROOT(T,S)$ is a Q-node then
13         replace the full children of ROOT$(T,S)$ and their descendants by $T(S',S')$
14     else
15         replace ROOT$(T,S)$ and its descendants by $T(S',S')$
16     $U \leftarrow (U-S) \cup S'$
17 return true
```

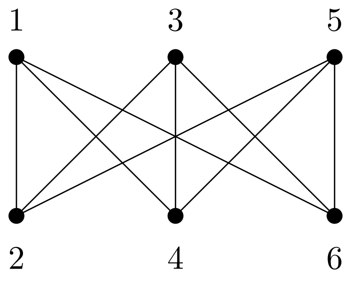


Figure 3.1: The s,t-numbering for $K_{3,3}$ that we use for our example.

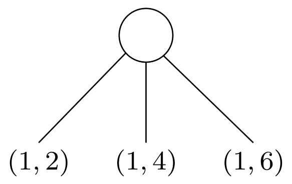


Figure 3.2: The initial PQ-tree with the outgoing edges of vertex 1 as leaves.

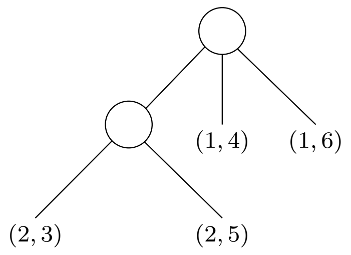


Figure 3.3: The PQ-tree after replacing $(1,2)$ with the new PQ-tree for the outgoing edges of vertex 2.

This step is then repeated at vertex 3. The procedure here is the same as in the previous iteration since vertex 3 also has just one incoming and two outgoing edges (for the resulting tree, see Figure 3.4).

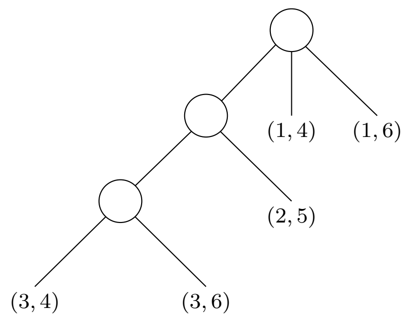


Figure 3.4: The PQ-tree after processing vertex 3.

The next iteration of the loop is the first one which is interesting as vertex 4 has two incoming edges. Thus, we need to make $(1, 4)$ and $(3, 4)$ consecutive in the current tree. Therefore, we need to introduce a Q-node that has nearly all the leaves as its children (see Figure 3.5).

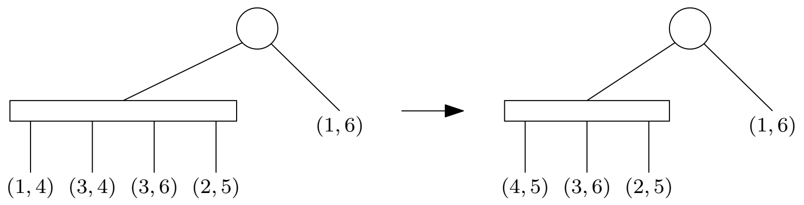


Figure 3.5: On the left, the PQ-tree after making the incoming edges of vertex 4 consecutive, and on the right, the replacement of the full children of the Q-node with the PQ-tree for the outgoing edges of vertex 4.

In the last iteration, the non-planarity of the graph is discovered. We process vertex 5, therefore, we need to make its incoming edges, $(2, 5)$ and $(4, 5)$, consecutive. But since we only are allowed to reverse the order of the leaves of the Q-node in Figure 3.5, there is no possible permutation for this, the reduction yields the empty PQ-tree and the algorithm terminates, returning false.

3.2 The algorithm by Boyer and Myrvold

The BoyerMyrvold planarity algorithm by edge addition from [BM04] is a further improvement on the algorithm that was introduced by Myrvold and Myrvold in 1999 [BM99]. The basic idea of the algorithm is from the planarity test by Booth and Lueker [BL76], which was described above, but instead of working with a PQ-tree or a PC-tree like the ShihHsu planarity test [SH99], the algorithm by Boyer and Myrvold relies on graph constructs. Here, the merging of biconnected components at cut vertices represents the operations that are done with a PC-tree. The improved version in [BM04] simplifies the check for planarity even further by focusing more on the addition of a single edge, so the detection of non-planarity can happen more fine-granularly. Thus, a possible planarity obstruction can be found more quickly.

3.2.1 The basic idea of Edge Addition

The goal of this algorithm is to be able to add each edge of the graph into an embedding without loosing its planarity. Before an edge is added, the embedding consists of a set of biconnected components that are separated through cut vertices. Adding an edge could then mean that two or more biconnected components are combined into one larger biconnected component (see Figure 3.6). In this example, the graph consists of two biconnected components that are connected through a cut vertex c, thus removing c would disconnect the graph. Now we want to add an edge (m, n) where m and n are located in two different biconnected components. By doing so, the two biconnected components are combined into one as the removal of c no longer disconnects them. In general, there are different ways on how we can add an edge. However, some of them might prevent us from retaining planarity since we potentially remove a vertex, let us call it w, from the external face that needs to be connected with another vertex on the external face in a later step (see Figure 3.6 (a)). If we for example need to add the edge (w, x), where x is on the external face of another biconnected component, this is not possible without crossing another edge causing the loss of the graph's planarity. But if we *flip* the biconnected component that contains w, which means that we reverse the order of the edges of the biconnected component's root that are adjacent to vertices in that biconnected component, w stays on the external face when adding (m, n) in that way and therefore (w, x) can be added without causing any problems (see Figure 3.6 (b)). So when adding an edge, the algorithm needs to make sure that all the vertices that are involved in later edge additions stay on the external face. These vertices are called *externally active.* In this context, a vertex is *pertinent*, if there is a back edge to the currently processed vertex that is not yet embedded or if there is a child biconnected component in the embedding that has a pertinent vertex. A biconnected component is called *pertinent* if it contains a pertinent vertex. Furthermore, a biconnected component is *externally active* if it has at least one externally active vertex. On the other hand, we call a biconnected component *internally active* if it is pertinent but has no externally active vertices. The same holds for inactive vertices. Finally, if a vertex or biconnected component meets none of those requirements, they are called *inactive.*

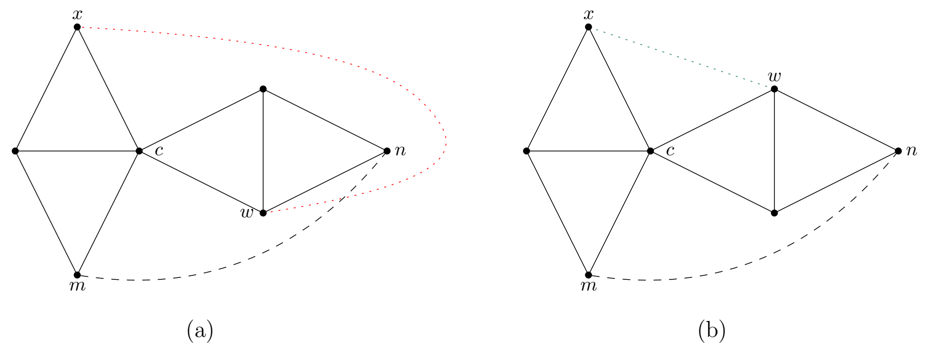


Figure 3.6: (a) By adding (m, n) like this, w is not on the external face anymore and, therefore cannot be connected to x without having to cross another edge. (b) By flipping the right component, w stays on the external face and can be connected with x without losing planarity.

3.2.2 Key operations

To test a graph's planarity using the technique of edge addition, Boyer and Myrvold described an algorithm [BM04], which consists of a preprocessing phase that performs some basic operations, a processing phase to determine the input graph's planarity, and a postprocessing phase to build the embedding if the input graph is planar or to provide the obstructing subgraph in case of non-planarity (see Algorithm 3.2).

Algorithm 3.2: BoyerMyrvoldEdgeAdditionPlanar

```
Input: Graph $G$
Output: Boolean representing whether the input graph is planar or not and
        either a planar embedding or a Kuratowski subgraph

// Preprocessing phase
1  $DFS(G)$
2  Lowpoint calculations for $G$
// Main processing phase
3  Creation and initialization of the embedding $\tilde{G}$ based on $G$
4  for vertex $v$ with $dfi = n - 1$ to $dfi = 0$ do
5      for Vertex c: DFS children of $v$ do
6          Embed tree edge $(v^c, c)$ as biconnected component
7      for Edge $(v, w)$: back edges incident to $v$ and a descendant $w$ do
8          Walkup$(\tilde{G}, v, w)$
9      for Vertex c: DFS children of $v$ do
10         Walkdown$(\tilde{G}, v^c)$
11     for Edge $(v, w)$: back edges incident to $v$ and a descendant $w$ do
12         if $(v^c, w) \notin \tilde{G}$ then
13             IsolateKuratowskiSubgraph$(\tilde{G}, G, v)$
14             return NONPLANAR, $\tilde{G}$
// Postprocessing phase
15 RecoverPlanarEmbedding()
16 return PLANAR, $\tilde{G}$
```

3.2.2.1 DFS and Lowpoint Calculations

The initial step of the algorithm is to compute a DFS tree. Boyer and Myrvold observed that the following property that is used in PQ-tree-based planarity algorithms also exists analogously for DFS trees. As seen earlier, with PQ-trees, a property of s, t-numbered Graphs can be used which makes it possible to embed the first k vertices in a way that the remaining vertices can be embedded in a single face. In the DFS tree, one can easily see that every vertex has a path of vertices with a lower DFI that leads to the DFS tree root. Therefore, the planarity algorithm processes the vertices in a reversed DFI order, so that when processing a vertex, there is always a path of unprocessed lower-numbered DFS ancestors that leads to the DFS tree root. This means that all the unprocessed vertices have to appear on a single face in the partial embedding, without loss of generality, the external face. To keep track of those vertices that need to be on the external face, we need also the *least ancestor* of each vertex, which is the vertex with the lowest DFI that can be reached via a back edge, and the *lowpoint*, which is the DFS ancestor with the lowest DFI that is reachable through a back edge from a descendant [BM04].

3.2.2.2 Initialization of the Embedding

After the preprocessing, the algorithm builds a new graph from scratch that will then be used for embedding the edges. As a starting point, each DFS tree edge is embedded into its own biconnected component, connecting the DFS parent with its DFS child. Since a vertex can have multiple DFS children, we need corresponding copies of the parent, which are denoted with v^c where v is the DFS parent and c a DFS child of v (see Figure 3.7). When this is done, we can start with the embedding of the back edges which is the part of the algorithm where it is decided whether the input graph is planar or not.

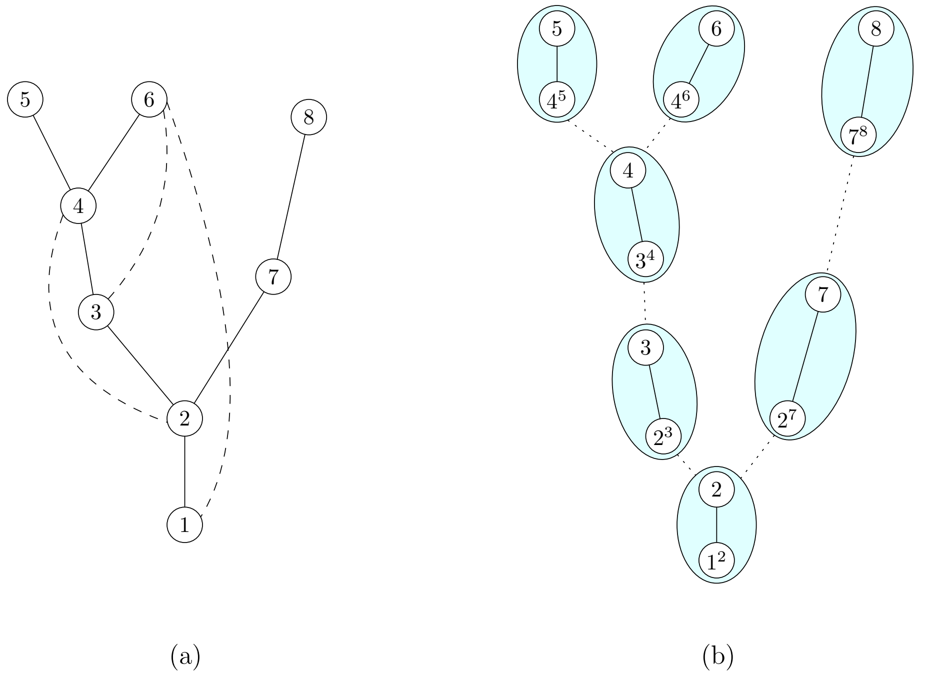


Figure 3.7: (a) A DFS tree for an arbitrary input graph. The dashed lines represent back edges. (b) The initial embedding where the tree edges of the DFS tree from (a) are already embedded. The initial biconnected components that result from this are marked blue. The root vertices are denoted with v^c, where c is a DFS child of v

.

3.2.2.3 The Walkup Subroutine

If we now want to add a back edge to the embedding, as mentioned earlier, two or more biconnected components might be merged into one due to the newly added edge. Let v denote the vertex whose incoming back edges are to be added. Therefore, we could just do a DFS on the DFS subtree of v to find all the descendant endpoints of the back edges. However, this would lead to a quadratic runtime in total since this DFS would have to be performed for every vertex with back edges. To maintain a time complexity of $O(n)$, we must limit the search only to the pertinent subgraph. The pertinent subgraph is the part of the graph that contains all the biconnected components that are involved in the merging process when adding the back edges to v. To be able to identify this subgraph, Boyer and Myrvold introduced a subroutine called Walkup, which is executed once for each back edge. If we want to add a back edge (w, v), the Walkup starts at w by marking it as pertinent (since the descendant endpoints of back edges are involved in the merging process in any case). Then the traversal of the external face starts to find the root of the current biconnected component. As there are two ways to traverse the external face (clockwise and counterclockwise), we do not know in advance which direction provides the shortest path to the root. If the wrong choice was made, maybe every vertex on the external face needs to be visited (which can be an arbitrary number of vertices), whereas the other direction just would have needed one step. To avoid this, when the traversal begins at the entry point of a biconnected component, a simultaneous traversal in both directions is started that stops as soon as the root has been found by one of the traversals. This way, we need at most two times the steps of the length of the shortest path to the root, which is sufficient. Once the root is found, it is recorded in its non-root copy so we know that this root is pertinent when adding the back edge in a later step. This non-root vertex is then taken as the entry point of the next biconnected component, where the same procedure is repeated until the starting vertex v is found. Additionally, each vertex that was visited during the traversal is marked as visited (by v) so that when multiple back edges need to be added to v, we know that we can stop at a certain vertex since the remaining pertinent root vertices have already been recorded by a previous run of Walkup. An illustration of how the Walkup procedure works can be seen in Figure 3.8.

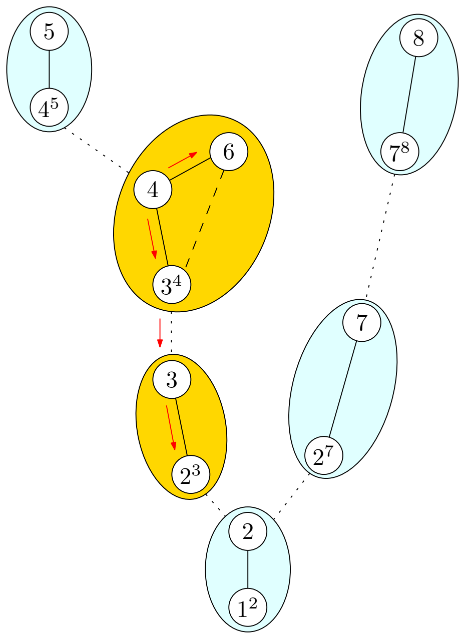


Figure 3.8: The Walkup on the DFS-tree from Figure 3.7. Currently, vertex 2 is processed, thus the back edge $(6, 3)$ is already embedded. The Walkup is started for the back edge $(4, 2)$. The simultaneous traversal begins at vertex 4 and finds vertex 3^4, then ascends to vertex 3, finds vertex 2^3 where it finishes. The biconnected components of the root vertices that were recorded are marked orange.

3.2.2.4 The Walkdown Subroutine

Now that we know all the biconnected components and cut vertices that are relevant for adding the incoming back edges of v, we can start the embedding process. To cover all of them, a run of the Walkdown is done for each DFS child of v. As seen earlier, there are again two directions to traverse the external face, so the Walkdown does both of them iteratively. Each of the traversals ends when a certain stopping condition is met, which we will cover later. Starting at v^c, the traversal goes along the vertices of the external face until a vertex is found that has a pertinent child biconnected component. Then, the procedure proceeds to the root of this vertex and continues. This is repeated until the endpoint w of the back edge to be embedded is found. Before adding that edge, all the biconnected components on the traversed path must be merged. This merging process might also include the flipping of some components as mentioned earlier. A biconnected component has to be flipped if

the traversal direction changes when entering the component.
A traversal is stopped if it gets back where it came from, thus vertex v^c, or if it crosses a non-pertinent externally active vertex, which Boyer and Myrvold called a *stopping vertex*, as explained in Section 3.2.1. If we were to proceed the Walkdown over the stopping vertex, after adding the back edge, it would not be on the external face anymore, and therefore it would not be possible to add other back edges to this vertex. To avoid this, the traversal is stopped and the other direction is tried by flipping the biconnected component to keep the externally active vertex on the external face but maintain the traversal direction. If the second run also encounters a stopping vertex, we have found proof of non-planarity since there is no way to avoid the crossing of the current back edge and a back edge that needs to be embedded in a later step. In this case, the algorithm returns NONPLANAR.
Additionally, when the traversal visits a vertex w that has more than one pertinent child biconnected component, the internally active child biconnected components need to be descended to before the externally active child biconnected components to make sure that embedding all the back edges that can be embedded without losing planarity is possible (since the Walkdown visits every vertex of an internally active biconnected component and comes back to w, but if we go to the externally active biconnected component, we will not get back to w after visiting the first externally active vertex). An example for the Walkdown process can be seen in Figure 3.9.

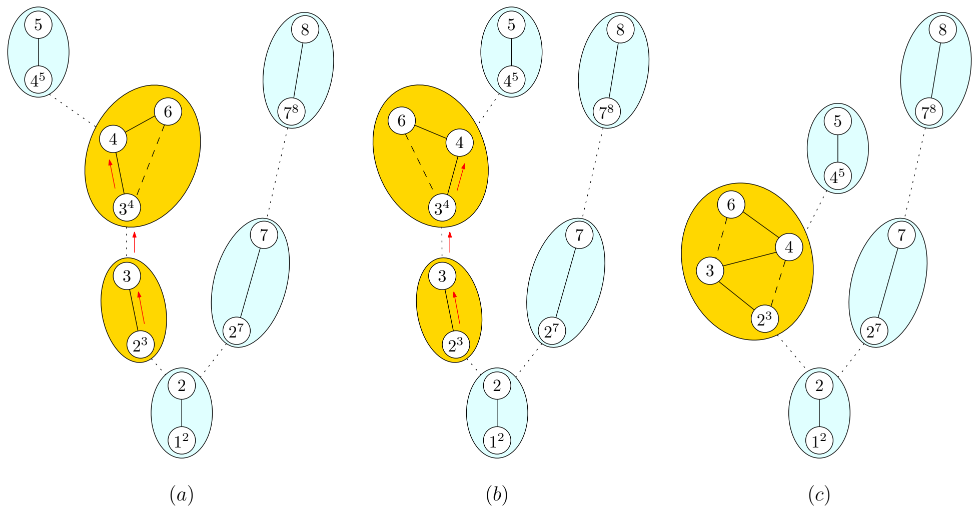


Figure 3.9: The Walkdown following the Walkup of Figure 3.8. (a) To demonstrate the flipping of a biconnected component, we start the Walkup traversal at vertex 2^3 in a counterclockwise direction. We go to vertex 3 and descend to the next biconnected component, rooted by vertex 3^4. The traversal now would go to vertex 6 but this is not allowed since it is a stopping vertex (vertex 6 is an endpoint of the back edge $(6, 1)$, see Figure 3.7). So we go directly to vertex 4 instead. (b) As we switched the traversal direction at vertex 3^4, this biconnected component must be flipped. (c) The back edge $(4, 2)$ can be embedded, and the two biconnected components are merged.

3.2.2.5 The Postprocessing Phase

As mentioned above, the Walkdown stops as soon as there is a back edge that cannot be added. This means that the whole input graph has no planar embedding. Therefore, a *Kuratowski subgraph isolation* is performed, which isolates the K_5 or $K_{3,3}$ subgraph that obstructs the planarity which is then returned. If every back edge of the input graph was successfully added, the input graph has a planar embedding which is recovered and also returned.

3.2.3 The Data Structure

To be able to perform all the operations in the required time complexity, a specially designed data structure is needed which is described in [BM04]. The vertices and edges are managed in one array as numerical indices where the vertices are stored in the front and the edges are stored at the back. The adjacency lists are realized as doubly linked cyclic lists where each vertex and edge has a pointer to a predecessor and a successor (see Figure 3.10). This makes it possible to go from vertex to vertex in the adjacency list in constant time. The same holds for deleting or adding edges to the adjacency list or concatenating the adjacency lists of two vertices. A similar structure is used for lists that are needed for some parts of the algorithm, where a single array is used to hold multiple doubly linked circular lists. Since it is assumed that in one such *list collection* no element is in more than one list at the same time, each list is represented as its head from which we can access the successor or the predecessor (the tail of the list) to iterate over the list in the corresponding direction.

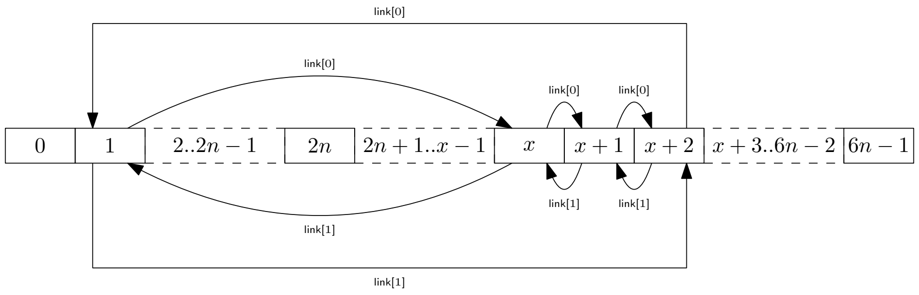


Figure 3.10: The array where the vertices and edges are stored. The real vertices can be found from 0 to $n-1$, its root copies from n to $2n-1$ and from $2n$ to $6n-1$. Each vertex and edge has two pointers link[0] and link[1] which are used to cyclically go through the adjacency list of a vertex either in a clockwise or counterclockwise direction. Here, vertex 1 has the adjacency list entries x, $x+1$ and $x+2$ with $2n < x < 6n-2$.

The vertices and edges themselves need to store some important information for themselves, like their DFI or a list of DFS children. The graph itself manages the collection of vertex and edge data as well as some other lists and flags. A listing of all the attributes can be found in the appendix of [BM04].

3.2.4 OGDF implementation

A C reference implementation is provided by Boyer and Myrvold (which is also included in the comparison to see how big the impact of the OGDF data structure is) but to be able to compare this algorithm to the others, we ported the algorithm to the OGDF library to use the graph data structures from the OGDF instead of the proprietary data structure that was used in the reference implementation. Although, the algorithm from [BM04] is already implemented in the OGDF [CGJ+13] but is a bit inefficient as we will see in Section 4.3, potentially due to some deviations from the reference implementation. Therefore, we tried to stay as close as possible to this data structure inside the OGDF to get the most out of the algorithm.

To emulate the coexistence of nodes (vertices in the naming scheme of the OGDF) and their root copies, which is difficult in general with the OGDF nodes, we defined a bNode as a pair of a node and a boolean to be able to tell if we are working with the real vertex or its root copy. Furthermore, to store all the information that every vertex and edge needs, we used NodeArrays and AdjEntryArrays to attach this information, which provides constant time access to the information of a given node or adjEntry. Also, we are using adjEntries to model the directed edges from the original data structure. To be able to alter the target of an adjEntry without modifying the input graph, we defined an additional type bEdge, which is a pair of an adjEntry and a bNode. An unaltered adjEntry contains an empty bNode (nullptr as node) as the target. An additional function checks if the target is not an empty bNode. In that case, the target is returned, the original target of the adjEntry otherwise. For the adjacency lists at the individual vertices, which stores bNodes, a list structure provided by the OGDF library was used as it provides $O(1)$ concatenation which is needed for merging vertices and was done by changing a pointer in the node-edge-array in the original implementation (what was not possible in the OGDF data structure). For the pertinentBicompList, the separatedDFSChildList, and the fwdArcList, lists from the C++ standard library in combination with storing iterators for the elements in those lists were used as they provide $O(1)$ deletion of elements without invalidating the iterators of the other elements of the list from which an element was deleted.
In contrast to the reference implementation, our implementation only tests for planarity, so the construction of a planar embed or a *Kuratowski* subgraph isolation for non-planar graphs is not included.

3.3 The algorithm by Haeupler and Tarjan

The last algorithm that we cover in this comparison was introduced by Haeupler and Tarjan in 2008 [HT08] and is one of the most recent ones, combining ideas and techniques already used by other algorithms like Lempel, Even, and Cederbaum [LEC67], Shih and Hsu [SH99], and Boyer and Myrvold [BM99, BM04]. It is also based on the PQ-tree data structure but unlike BoothLueker, no s,t-numbering is needed. Instead, it uses a DFS (therefore, the input graph does not need to be biconnected, respectively there is no need to isolate a graph's biconnected components).
Furthermore, a small modification to the PQ-tree data structure was made by introducing a so-called *special leaf*, which is attached to the root of a PQ-tree and changes the tree to maintain circular orders with the special leaf as the first element. If we perform a reduction on this modified PQ-tree where the special leaf is not included in the set on which the reduction is performed, the reduction process is the same as described by Booth and Lueker [BL76]. If it is included though, the tree is reduced with the complement of the set, therefore this is called a *complement reduction*.
As mentioned, the algorithm uses a DFS to process each vertex of the input graph. If the graph is not connected, the following routine is performed on each of its connected components. The planarity check runs *on the fly* with the DFS, thus it is done during the DFS traversal of the graph and not by some additional routines performed after the DFS. The basic idea is to have a PQ-tree for each DFS ancestor v of the currently processed vertex, each with a set $S(v)$ of leaves, which eventually will be combined when walking back the DFS tree. Therefore, when traversing a tree edge (v, w), where v is the current and w a newly discovered vertex, we create a new PQ-tree that consists of the special leaf for (v, w) and its single child for v, which is a P-node. Thus, $S(v)$ contains only (v, w). If a back edge (v, w) is found, it becomes a new child of the P-node for v and is added to $S(w)$. Lastly, when no more new vertices or other back edges can be found on a vertex, the DFS ascends back to its DFS parent over a tree edge (v, w). Here, the PQ-tree for w is reduced to make the elements in $S(v)$ consecutive. If we get the empty PQ-tree from

this reduction, the graph is not planar and the algorithm terminates. If not, all elements in $S(v)$ are removed from the reduced tree as are all the P and Q-nodes without children. Finally, we add this tree to the P-node of v (except if v has a DFI of 0).

We remark that the implementation that we included in our comparison uses PC-trees instead of PQ-trees which are a similar data structure but simpler and as was observed, more efficient [FPR23]. For embedding, we used the bucket embedding method [Feu23] that was also provided with the implementation.

3.3.1 An Example for the HaeuplerTarjan Algorithm

Since the description of the algorithm is rather abstract, we provide another small example to demonstrate how the algorithm works. For this, we use the connected graph from Figure 3.11.

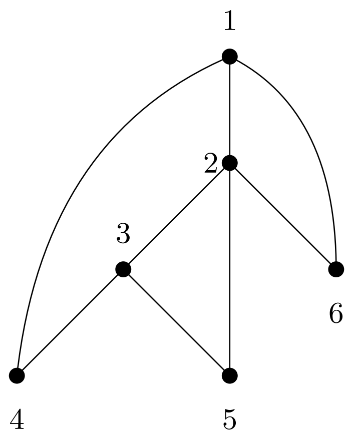
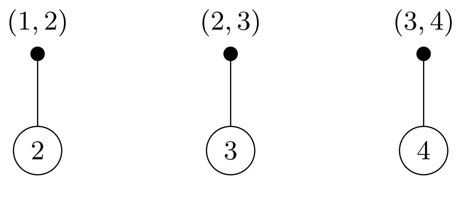


Figure 3.11: A small planar connected graph. The vertices have already been numbered in the order they will be processed by the DFS.

The first steps of the DFS cover visiting the vertices 1, 2, 3, and 4. According to the description, three PQ-trees are created for the latter three and the leaves are added accordingly to the sets (see Figure 3.12). Notice that we only show $S(1)$, $S(2)$, and $S(3)$ since the sets for the other vertices are always empty in this example.

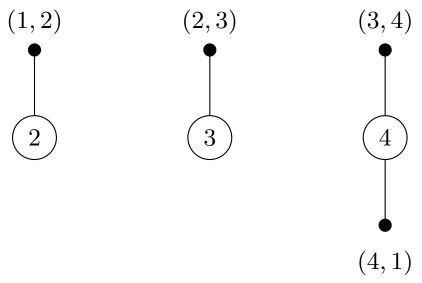


$S(1)$	$\{(1,2)\}$
$S(2)$	$\{(2,3)\}$
$S(3)$	$\{(3,4)\}$

Figure 3.12: The initial PQ trees and the sets of their leaves when descending the DFS tree to 4

When the DFS reaches vertex 4, no new vertex can be discovered, so the back edge $(4, 1)$ is found. It is added to the P-node for 4 and to $S(1)$ (see Figure 3.13).

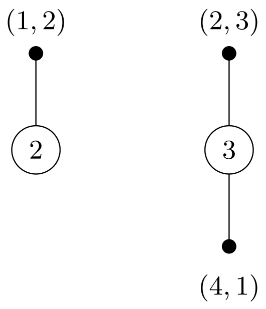
Since this was the only back edge of vertex 4, we go back on the tree edge $(3, 4)$. Therefore, we make $S(3) = \{(3, 4)\}$ consecutive on the PQ-tree of vertex 4. Since this is only one edge, the reduction is trivial and we simply remove $(3, 4)$ from the tree and the set (see



$S(1)$	$\{(1,2),(4,1)\}$
$S(2)$	$\{(2,3)\}$
$S(3)$	$\{(3,4)\}$

Figure 3.13: Finding the back edge $(4, 1)$.

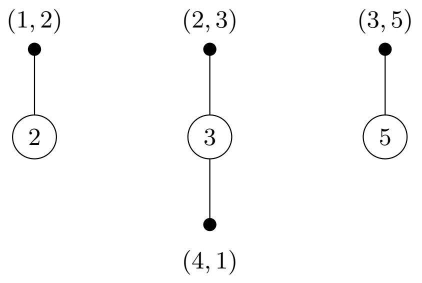
Figure 3.14).



$S(1)$	$\{(1,2),(4,1)\}$
$S(2)$	$\{(2,3)\}$
$S(3)$	$\{\}$

Figure 3.14: Going back to vertex 3 via tree edge $(3, 4)$.

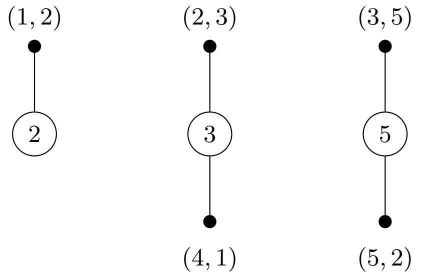
From vertex 3, we then discover vertex 5 and also create a PQ-tree accordingly (see Figure 3.15).



$S(1)$	$\{(1,2),(4,1)\}$
$S(2)$	$\{(2,3)\}$
$S(3)$	$\{(3,5)\}$

Figure 3.15: The DFS discovers vertex 5.

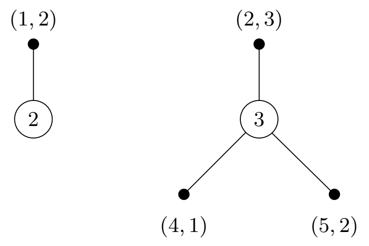
At vertex 5, we find the back edge $(5,2)$, which is then also added to the PQ-tree for vertex 5 and $S(2)$ (see Figure 3.16).



$S(1)$	$\{(1,2),(4,1)\}$
$S(2)$	$\{(2,3),(5,2)\}$
$S(3)$	$\{(3,5)\}$

Figure 3.16: Finding the back edge $(5,2)$.

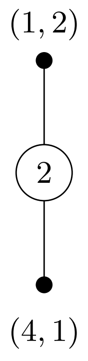
Then, vertex 5 is also finished and we go back to vertex 3 over the tree edge $(3,5)$. We make $S(3)$ consecutive and after removing $(3,5)$ from the tree, we attach it to the P-node of the tree for vertex 3 (see Figure 3.17).



$S(1)$	$\{(1,2),(4,1)\}$
$S(2)$	$\{(2,3),(5,2)\}$
$S(3)$	$\{\}$

Figure 3.17: Going back to vertex 3 via tree edge $(3,5)$.

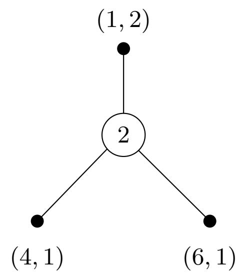
We continue traversing upwards the DFS tree and go back to vertex 2. Again, we reduce the tree for vertex 3 analogously to Figure 3.16 and attach it to the tree of vertex 2 (see Figure 3.18).



$S(1)$	$\{(1,2),(4,1)\}$
$S(2)$	$\{\}$
$S(3)$	$\{\}$

Figure 3.18: Going back to vertex 2 via tree edge $(2,3)$.

Now, we have finished the left subtree of vertex 2. The other subtree works the same (for the result after coming back again to vertex 2, see Figure 3.19).



$S(1)$	$\{(1,2),(4,1),(6,1)\}$
$S(2)$	$\{\}$
$S(3)$	$\{\}$

Figure 3.19: The resulting PQ-tree after the right subtree of vertex 2 has been traversed by the DFS.

The final reduction on the tree does not make any further changes since $S(1)$ are exactly the leaves. Vertex 1 was the starting point of the DFS and has no more undiscovered neighbors, therefore, the algorithm finishes and returns `true`.

4. Evaluation and Comparison

In this chapter, we do an evaluation and comparison of the algorithms explained above. In the following sections, an overview of the test data set on which the algorithms operate and the testing setup is given. Finally, the runtime of these algorithms on the test data set is measured to get a comparison of the algorithms to find out which of them works best in general or in specific situations.

4.1 Test Data

To compare the runtime of the algorithms, we need some graphs on which we can run the algorithms. First, we need to decide which graphs are significant to get a meaningful result. Each of the algorithms works differently, so when the input graph is non-planar, it could be that one algorithm finds the planarity obstruction in one of the first steps while another algorithm needs to process nearly the complete input graph until the outcome is fixed. Thus, when using non-planar inputs, comparing the runtimes does not give us a good comparison. This means that we mainly concentrate on planar graphs as our test data, although, some non-planar graphs will be included.
For the test data set of planar graphs, without loss of generality, only connected graphs are used (since the question of planarity for non-connected graphs can be reduced to the planarity of its connected subgraphs). The algorithms are tested with different types of graphs to see if some algorithms work very well or rather slowly for a specific type of graph. The types that are considered are connected, biconnected, and triconnected graphs respectively graphs with different densities. For this, the OGDF graph generator takes a given number of vertices and edges and builds a random connected planar graph from it. For higher densities, the graph then might be biconnected or even triconnected. Thus, when talking about connected graphs, we mean a graph that is connected but not biconnected. The same holds analogously for biconnected graphs.
For the following, let n denote the number of vertices of the graph. To get a wide range of graphs, we used three different densities for the input graphs, starting with $n - 1$ edges. This is the smallest number possible to get a connected graph and will therefore result in a tree. The other densities we used are $2n$ to get graphs with medium density and $3n - 6$, the maximal number of edges a planar graph can have. Also, due to this high density, all of those graphs are triconnected. We also decided to include some grid graphs since they are also planar and make our test data set even more diverse.
For the number of vertices, we started with $n = 1000$, since the difference in runtime is

not that significant for smaller instances. We went up to $n = 1000000$ vertices to also get some large instances. The number of vertices is increased in steps of 5000 to keep the total number of graphs within a limit but still gives enough detail to spot potential runtime changes at a certain graph size. For each graph size, three different graphs were generated. With this, unexpected runtime deviations are avoided, which could happen if a graph is disproportionately complicated for its size. The grid graphs start with a size of 100×100 and go up to 892×892. Furthermore, as we found out after some test runs, the random connected planar graphs we generated with the densities mentioned above were in our case always connected or triconnected, but never biconnected. Thus we generated another set of graphs with $2n$ edges and the specifications from above, but this time with the condition of being biconnected instead of just connected. For the non-planar instances, we started by generating a random connected planar graph with $2n$ edges. Then we added edges one after the other until the graph was no longer planar (which was tested after each step by the BoothLueker algorithm of the OGDF. This leaves us with a total number of 3582 test graphs.
To generate the test data, we used the built-in graph generation functionality of the OGDF, which provides a function to generate random connected planar graphs from a given number of vertices and edges. The graphs are stored as .gml (*Graph Modelling Language*) files since it is the format the OGDF uses for reading graphs. The graph type is computed by the graph generator after a graph has been generated and is then stored in the label of the node with index 0, so when the graph is read again, the algorithm knows the graph type to print it in the output string.
Since the reference implementation by Boyer and Myrvold is also included in the comparison, the .gml files must be additionally converted to a format that can be read by the implementation, which is a simple adjacency list in a plain-text file where the first line contains the number of vertices, followed by each vertex and its adjacent vertices. Each line (except the first one) ends with a -1 so the end of a vertex's list can be determined. We also made a small modification to this graph import format by storing an integer in the last line below the last vertex, which represents the graph class. We implemented a small tool in our testing framework to do a proper conversion.

4.2 Testing Setup

Our testing framework consists of multiple executables, each of them containing one of the algorithms. They take a path to a graph file (.gml or .txt) as their program argument and produce a single output on the console, which consists of the name of the algorithm, the filename of the graph, whether the input graph was planar, if a planar embedding was created, the graph class, the number of vertices, the number of edges, the sum of those two values, the vertex-edge ratio and the runtime. The output is formatted in CSV syntax. To verify that our implementation works correctly, we compared all its outputs from the entire test set with the outputs of the existing OGDF implementations. For the time measurement, we used the chrono library for the C++ implementations and the functionality provided by time.h for the C implementation since chrono is only available in C++. The duration of one run is captured by taking the system time when the actual computation began and when it ended, so the reading of the graph and the console output is not in the measurement. To achieve the highest possible degree of precision, the runtime is measured in nanoseconds. Each test graph is tested three times (with and without embedding) in succession to get a good median of a graph's runtime since it could happen that one run is unexpectedly slower for example due to external influences.
We run our algorithms on a *SLURM* cluster, so each graph is tested in a separate batch job. This saves us time since the jobs run in parallel on the nodes of the selected host. The file names of the test graphs have been stored in an additional text file before, so we

made a bash script to start a job for each line in this file. After all jobs have finished, the individual console outputs that contain the information explained above are stored in .out files by the *SLURM* cluster. Another bash script collects all the content from those files and appends them to a single CSV file. A simple Python script reads this CSV file and draws the plots from Section 4.3 using the seaborn [Was21] and the pandas [pdt20, WM10] library and saves them as a PDF file.
The tests were executed on an Intel Xeon E5-2690v2 (10 cores with 3.00 GHz each). All our algorithms are sequential, they are executed on a single core. The cluster has 17 nodes (however, node 17 was excluded in our testing setup since it has a different CPU), each one with 64 GiB of RAM. 1 GiB of RAM was allocated for each job. The cluster runs Linux kernel version 6.1.0-18-amd64. The implementations were compiled with GCC version 12.2.0, using the optimization -03 -march=native -mtune=native. The OGDF library was used in version 2023.09 (except for the HaeuplerTarjan algorithm, for which we used the commit 7a95bd8946c6af6becdac08d31e1f9ee05affd42 due to compatibility issues) and was compiled using its Release build type.

4.3 Results

For a fine granular comparison, we split the evaluation of the test results into the graph classes and densities defined in Section 4.1. In the following graphs, our implementation is denoted with BoyerMyrvoldPort, the existing OGDF implementation with BoyerMyrvoldOGDF, and the C reference implementation with BoyerMyrvoldPure. As mentioned in Section 3.2.4, our implementation does not provide a planar embedding and is therefore not included in the plots for doing planar embeddings. It is difficult to directly compare BoyerMyrvoldPure to the other algorithms since it does not use the OGDF data structure but it is included in the plots so we can see how much the use of the OGDF impacts the efficiency. Since our BoyerMyrvold implementation does not provide a planar embedder, as mentioned in Section 3.2.4, it is not included in the plots that show the time for planar embedding. Also, note that the unit on the x-axis of the plots (number of vertices) is millions (10^6), except for the grid graph plots.

Figure 4.1 shows the runtimes of the algorithms on random planar graphs with a density of $m = n - 1$, respectively trees. We observe that BoyerMyrvoldOGDF is the slowest, followed by BoothLueker, which is about 15% faster. Our implementation of BoyerMyrvold is then about 35% faster than the OGDF implementation. HaeuplerTarjan and the reference implementation of BoyerMyrvold are by far the fastest, being about 80% faster than our implementation and as fast as the reference implementation of BoyerMyrvold. The reference implementation is about 70% faster than our implementation and about 80% faster than the existing OGDF implementation.
When we look at the embedding times, we see that HauplerTarjan has the highest slowdown (see Figure 4.2), while the impact on the others is rather low, the runtime increases only with a factor of about 1.2 with the BoyerMyrvold reference implementation tending be more unstable since there are also some jumps to a factor above 1.3. Interestingly, BoothLueker is even faster with embedding than without, which might be due to how it was implemented in the OGDF.

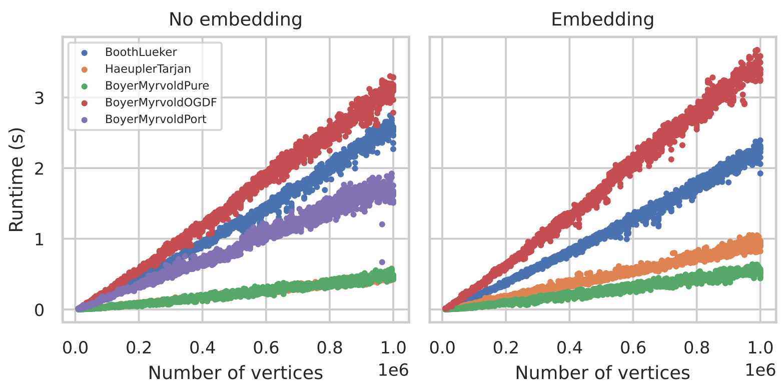


Figure 4.1: Runtime for random planar graphs with a density of $m = n - 1$ (trees).

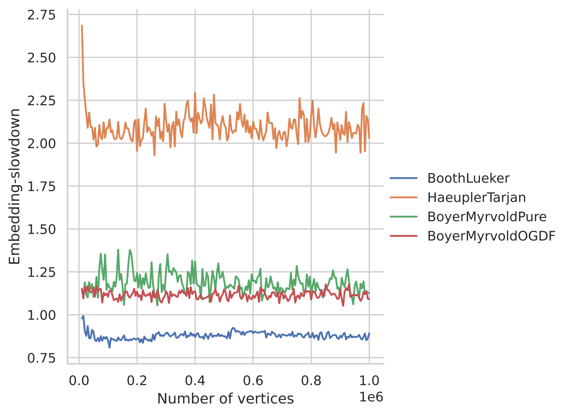


Figure 4.2: Performance decrease for additional embedding for random planar graphs with $m = n - 1$ edges.

For random connected planar graphs with $m = 2n$ edges, we see a similar result as for $m = n - 1$ edges (see Figure 4.3). Since for $m = 2n$, we have connected and biconnected graphs, we split them into two plots (see Figure 4.3). BoothLueker is now the slowest. For connected graphs, BoyerMyrvoldOGDF is now 25% faster than BoothLueker, while this is almost 30% for biconnected graphs. Our implementation is about 30% faster for connected graphs than BoyerMyrvoldOGDF. For biconnected graphs, this goes up to about 35%. HaeuplerTarjan is for connected graphs about 70% faster than our implementation, which also holds for biconnected graphs. The reference implementation of BoyerMyrvold is for both graph classes a bit faster than HaeuplerTarjan. The embedding slowdown is almost identical to the tree graphs with the exception that BoothLueker, besides still having the lowest performance impact, is as fast as without embedding for connected graphs and for biconnected graphs, it now has a similar slowdown as BoyerMyrvoldOGDF and BoyerMyrvoldPure (see Figure 4.4).

The runtimes for random maximal planar graphs again confirm the result of the previous graphs with BoothLueker being the slowest. For $m = 3n - 6$ edges, BoyerMyrvoldOGDF is about 35% faster while BoyerMyrvoldPort is about 35% faster than BoyerMyrvoldOGDF. HauplerTarjan again is the fastest OGDF-based implementation, being almost 70% faster than BoyerMyrvoldPort (see Figure 4.5). The embedding slowdown has not changed much from the previous graphs' densities but BoothLueker again has nearly no slowdown when creating a planar embedding (see Figure 4.6).

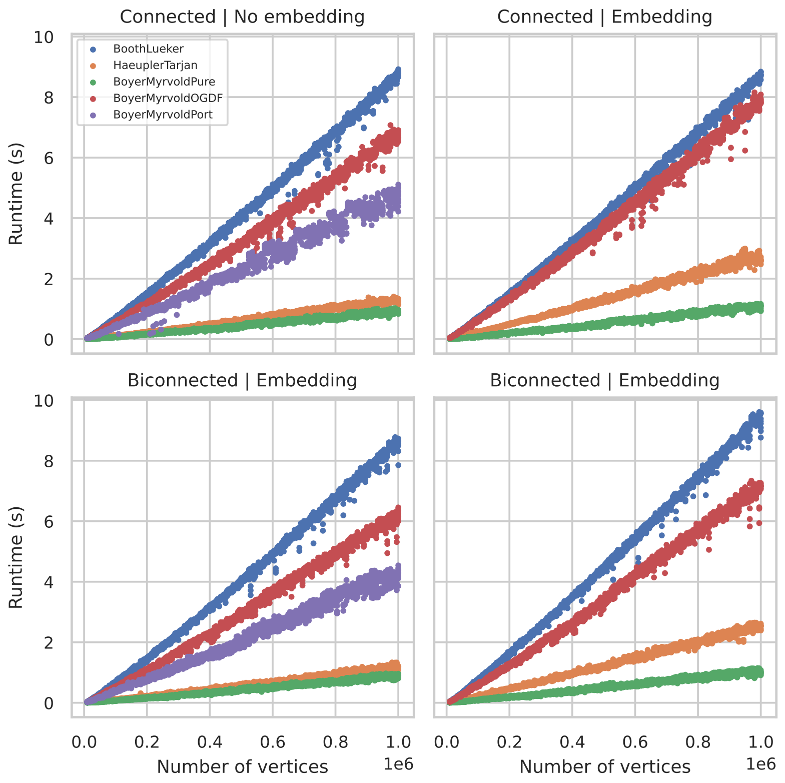


Figure 4.3: Runtime for random planar graphs with a density of $m = 2n$.

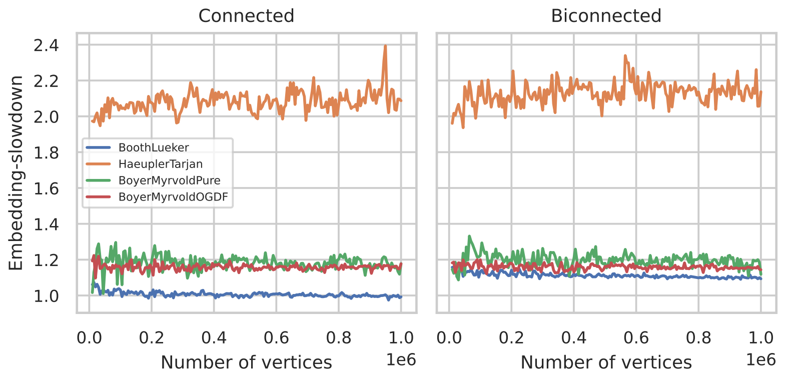


Figure 4.4: Performance decrease for additional embedding for random planar graphs with $m = 2n$ edges.

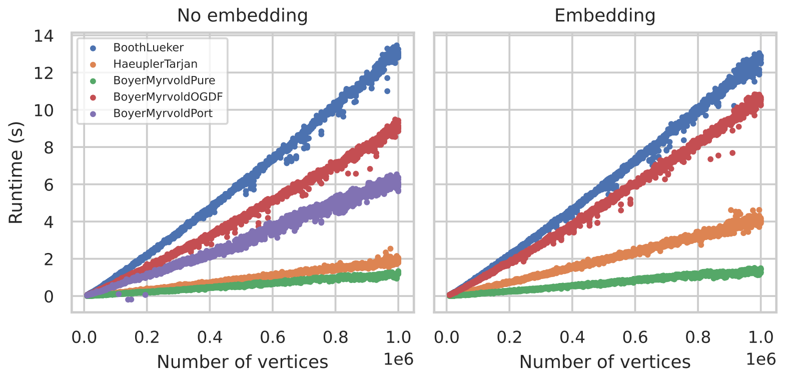


Figure 4.5: Runtime for random planar graphs with a density of $m = 3n - 6$.

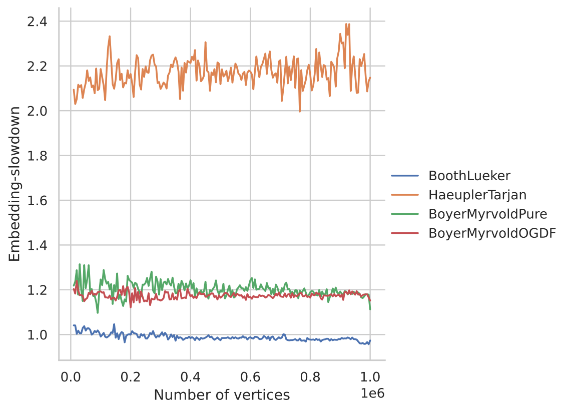


Figure 4.6: Performance decrease for additional embedding for random planar graphs with $m = 3n - 6$ edges.

We also did some plots for each of the graph classes but we noticed that our triconnected graphs are exactly the ones with $m = 3n - 6$ edges. Furthermore, since there are no biconnected graphs with $n - 1$ edges and as we did a split on graph classes in Figure 4.3, this graph class is also already covered. The same holds for connected graphs. Thus, these plots are not included here due to redundancy.
However, for the grid graphs, the results are quite interesting. Our implementation is the slowest, followed by BoyerMyrvoldOGDF. For both, the scattering is higher than for the other graph classes. Surprisingly, BoothLueker is here faster than BoyerMyrvoldOGDF and our implementation. The HaeuplerTarjan again is the fastest OGDF-based implementation but this time just a bit over 30% faster than BoothLueker. The gap between HaeuplerTarjan and the reference implementation of BoyerMyrvold is also much larger than in the previous comparisons (see Figure 4.7). In contrast, the slowdown by doing a planar embedding is lower than before for every implementation, except for BoothLueker where the runtime with embedding is now increased by a factor over 1.2 (see Figure 4.8).

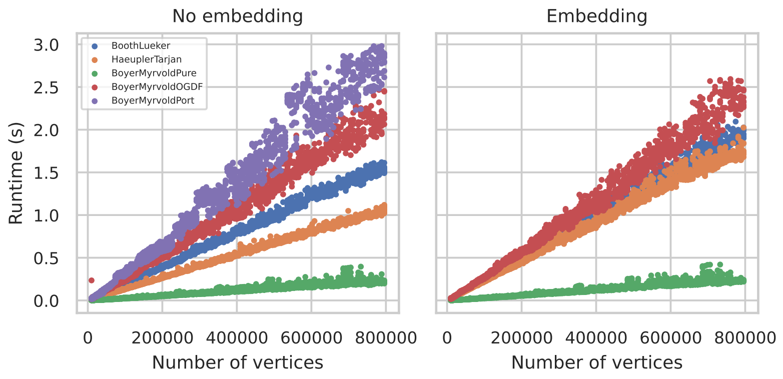


Figure 4.7: Runtime for grid graphs.

Figure 4.9 shows the runtimes for the non-planar graphs. Although these runtimes are not necessarily representative of the efficiency of the implementations, we see a similar picture as in the previous results (except the grid graphs).

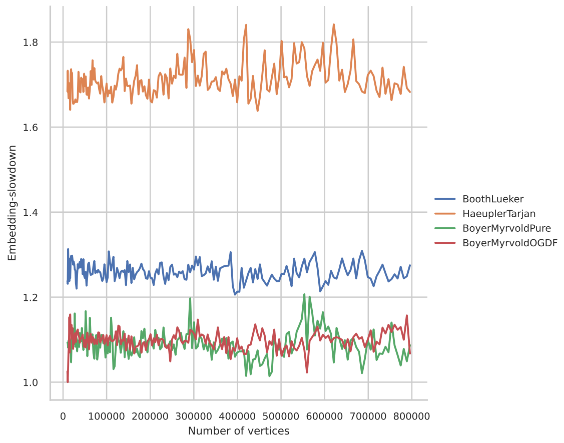


Figure 4.8: Performance decrease for additional embedding for grid graphs.

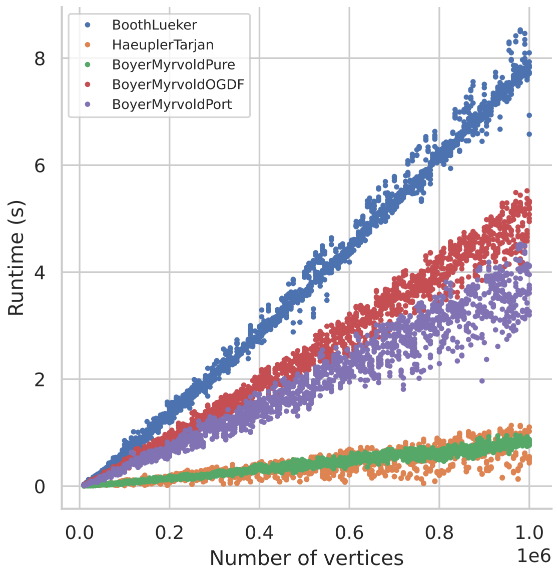


Figure 4.9: Runtime of the algorithms for non-planar graphs with at least $m = 2n$ edges.

As expected, the port of BoyerMyrvoldPure to the OGDF data structure slowed down the algorithm. Figure 4.10 shows how big the overhead of the data structure was. For $m = n - 1$ edges, the reference implementation is faster on average by a factor of 4. For less than $400,000$ vertices, this factor is even higher. For $m = 2n$ and $m = 3n - 6$ edges, the speedup factor is about 5, but a bit higher again for graphs with less than $200,000$ vertices.

As mentioned in Chapter 3, the existing OGDF implementation of the BoyerMyrvold algorithm suffers from some inefficiency, which was confirmed by our comparison. For $m = n - 1$ edges, our implementation is about 1.75 times, for $m = 2n$ and $m = 3n - 6$ about 1.5 times faster (see Figure 4.11). The peaks that can be seen for $m = 3n - 6$ at around $200,000$ vertices might be due to some unexpected deviation for some graphs.

Finally, Figure 4.12 shows the speedup of HaeuplerTarjan, which we observed to be the fastest OGDF-based planarity implementation available, in comparison to our implementation of BoyerMyrvold. For $m = n - 1$, we have a factor of a bit over 4, for $m = 2n$, the factor is

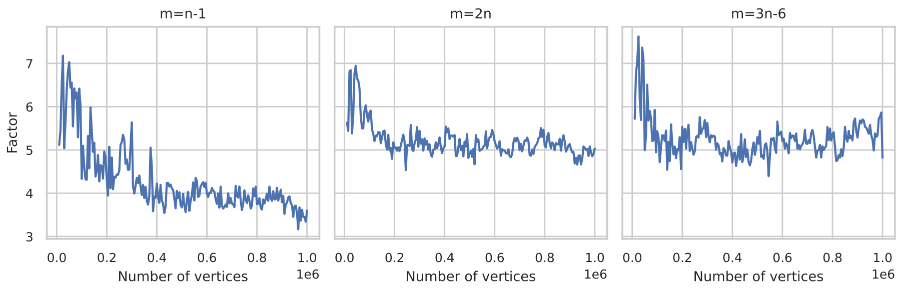


Figure 4.10: Speedup of the reference implementation of BoyerMyrvold compared to our implementation.

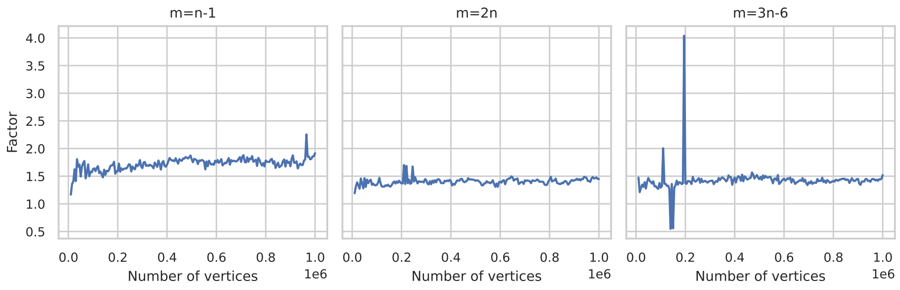


Figure 4.11: Speedup of our implementation of BoyerMyrvold compared to the OGDF implementation of BoyerMyrvold.

somewhere between 3.5 and 4 and for $m = 3n - 6$, it is between 3 and 3.5. So we see that the speedup decreases with increasing density of the graph.

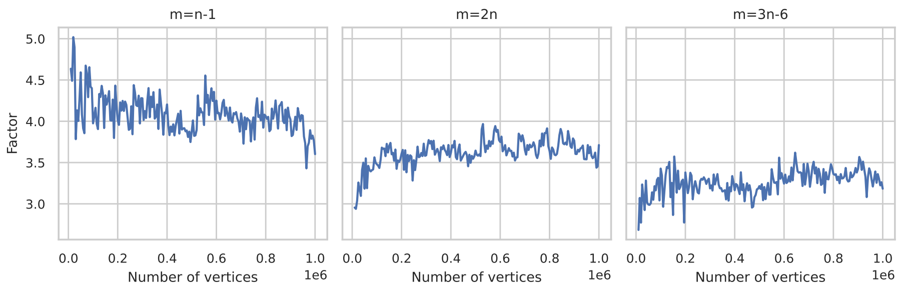


Figure 4.12: Speedup of HaeuplerTarjan compared to our implementation of BoyerMyrvold.

5. Conclusion

In this thesis, we introduced the basic concept of planar graphs and some important techniques and data structures that are closely connected to planarity. We gave an overview of existing planarity algorithms and did a description of the algorithms that we included in our comparison, namely the algorithms by Booth and Lueker [BL76], Boyer and Myrvold [BM04], and Haeupler and Tarjan [HT08]. There, we put the focus on the algorithm by Boyer and Myrvold since the existing implementation in the OGDF, which we used as the graph library to have a common data structure to compare the algorithms, has some potential efficiency issues. We described the individual steps of this algorithm in detail and provided our own OGDF implementation that emulates the reference implementation provided by Boyer and Myrvold as closely as possible to solve those issues.
We then measured the performance of those implementations of the algorithms described in Chapter 3 to evaluate and compare them. We observed, that in general, the newer algorithms are the fastest, namely the BoyerMyrvold and the HaeuplerTarjan, although for the latter, the impact of additionally creating a planar embedding is much higher than for the other tested implementations. In our tests, the implementation of the algorithm by Booth and Lueker was the slowest, followed by the OGDF implementation of the BoyerMyrvold algorithm and ours. The implementation of the HaeuplerTarjan algorithm was the fastest in our comparison. But we also got some deviating results considering grid graphs where BoothLueker was faster than the two BoyerMyrvold implementations. Overall, every implementation in our comparison provides a linear runtime.

There are other planarity algorithms that we did not include in our comparisons like the one by Shih and Hsu [SH99], which also uses PC-trees like the implementation of the HaeuplerTarjan algorithm since there was no OGDF implementation available. As shown by Fink, Pfretzschner, and Rutter [FPR23], the PC-tree data structure is faster than the PQ-data structure. Therefore an OGDF version of the BoothLueker algorithm using PC-trees would also be interesting to compare. Another area of planarity testing that would have to be explored is parallelism. There are some proposals for parallel algorithms by Klein and Reif [KR88], and Ramachandran and Reif [RR94] for which an overall performance analysis and comparison would be interesting too.

Bibliography

[AH74] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[AH77] K Appel and W Haken. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

[AHK77] Kenneth Appel, Wolfgang Haken, and John Koch. Every planar map is four colorable. part ii: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 1977.

[AP61] L. AUSLANDER and S. V. PARTER. On imbedding graphs in the sphere. *Journal of Mathematics and Mechanics*, 10(3):517–523, 1961.

[Bad64] Wilhelm Bader. Das topologische problem der gedruckten schaltung und seine lösung. *Archiv für Elektrotechnik*, 49:2–12, 1964.

[BKR15] Thomas Bläsius, Stephen G. Kobourov, and Ignaz Rutter. Simultaneous embedding of planar graphs, 2015.

[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 *SODA*, pages 140–146, 1999.

[BM04] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[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.

[DKT09] Zdeněk Dvořák, Ken-ichi Kawarabayashi, and Robin Thomas. Three-coloring triangle-free planar graphs in linear time. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1176–1182. SIAM, 2009.

[ET76] Shimon Even and Robert Endre Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2(3):339–344, 1976.

[FCE95] Qing-Wen Feng, Robert F. Cohen, and Peter Eades. Planarity for clustered graphs. In Paul Spirakis, editor, *Algorithms — ESA '95*, pages 213–226, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Feu23] Tim-Florian Feulner. Deriving embeddings and triconnectivity from the haeupler-tarjan planarity test, 2023.

[FPR23] Simon D. Fink, Matthias Pfretzschner, and Ignaz Rutter. Experimental comparison of pc-trees and pq-trees. *ACM J. Exp. Algorithmics*, 28, oct 2023.

[Gol63] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in the plan. In *John R. Edmonds Jr., editor: Graphs and Combinatorics Conference*, page 2 unn. pp. Princeton University, 1963.

[Gro59] H Grotzsch. Ein dreifarbensatz fur dreikreisfreie netze auf der kugel. *Wiss. Z. Martin Luther Univ. Halle-Wittenberg, Math. Nat. Reihe*, 8:109–120, 1959.

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

[HT08] Bernhard Haeupler and Robert E. Tarjan. Planarity algorithms via pq-trees (extended abstract). *Electronic Notes in Discrete Mathematics*, 31:143–149, 2008. The International Conference on Topological and Geometric Graph Theory.

[KM08] Vladimir P Korzhik and Bojan Mohar. Minimal obstructions for 1-immersions and hardness of 1-planarity testing. In *International Symposium on Graph Drawing*, pages 302–312. Springer, 2008.

[KR88] Philip N. Klein and John H. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Sciences*, 37(2):190–246, 1988.

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

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs: International Symposium: Rome, July, 1966*, pages 215–232. Gordon and Breach, New York, 1967.

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

[pdt20] The pandas development team. pandas-dev/pandas: Pandas, Feb 2020.

[Rin65] Gerhard Ringel. Ein sechsfarbenproblem auf der kugel. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 29, pages 107–117. Springer, 1965.

[RR94] Vijaya Ramachandran and John Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49(3):517–561, 1994.

[RSST97] Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas. The four-colour theorem. *journal of combinatorial theory, Series B*, 70(1):2–44, 1997.

[SH99] Wei-Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theoretical Computer Science*, 223(1):179–192, 1999.

[Was21] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

[WM10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.