Constrained Planarity Algorithms in Theory and Practice

Simon Dominik Fink

Universitätsdissertation
zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)

in der Wissenschaftsdisziplin
Theoretische Informatik

eingereicht an der
Fakultät für Informatik und Mathematik
der Universität Passau

Datum des Rigorosums: 12. Dezember 2023

This work is licensed under a Creative Commons License:
Attribution 4.0 International.
This does not apply to quoted content from other authors.
To view a copy of this license visit
http://creativecommons.org/licenses/by/4.0/

Betreuer

Prof. Dr. Ignaz Rutter
Universität Passau

Gutachter

Prof. Dr. Ignaz Rutter
Universität Passau

Prof. Dr. Jens M. Schmidt
Universität Rostock

Published online at the
Institutional Repository of the University of Passau:
https://doi.org/10.15475/cpatp.2024

Acknowledgments

Meine Promotionszeit war hauptsächlich von Corona und dem eingehenden Social Distancing mit Lehre, Besprechungen und Konferenzen via Zoom geprägt. Das hat die vier Jahre als Doktorand sicher nicht einfacher gemacht, dafür haben mich einige besondere Menschen durch diese Zeit begleitet und umso mehr unterstützt – dafür möchte ich Danke sagen.

Zuallererst danke ich meinem Doktorvater Ignaz Rutter, dass er mir gegen Ende meines Masterstudiums das interessante und facettenreiche Forschungsgebiet der Theoretischen Informatik gezeigt und es mir zugetraut hat, in dem mir bis dahin weitestgehend unbekannten Feld des Graphenzeichnens zu promovieren. Als Betreuer stand Ignaz' Tür immer offen für anregende Gespräche und Diskussionen und so war hilfreiches Feedback zum neuesten Aufschrieb des nächsten Beweises nie weit entfernt. Gleichzeitig trat er als sehr nahbarer Chef auf, der spontan auch mal im Mitarbeiter-Büro zum Prokrastinieren vorbei kommt oder abends auf ein Bierchen einlädt.

Als nächstes möchte ich meinen Kolleginnen und Kollegen danken: Matthias Pfretzschner, Miriam Münch, Patricia Bachmann, Peter Stumpf und Sandhya T. P. Besonders in meiner Anfangszeit hat Peter mir geholfen, mich mit den formellen Eigenheiten des Themengebiets zurechtzufinden. Später kam immer mehr Matthias dazu und damit eine Zusammenarbeit, auf der viele der praktischen Aspekte dieser Arbeit fußen. Egal wie die Massen-Mitarbeiter-Haltung im viel zu kleinen Büro aussah, ob zusammen mit Peter und Miriam oder später zusammen mit Matthias und Patricia eine Tür weiter, es herrschte immer eine sehr gute Stimmung und ich werde unsere gemeinsame Zeit vermissen. Ein herzliches Dankeschön geht auch an alle weiteren Ko-Autoren für die gute Zusammenarbeit an gemeinsamen Papern. Dass die Universität Passau so familiär und dadurch für mich über viele Jahre wie ein zweites Zuhause gewesen ist, verdankt sie den vielen herzlichen Menschen im Vorder- und Hintergrund. Auch dafür möchte ich Danke sagen und hoffe, dass sich die kommenden Generationen hier ebenso wohlfühlen wie ich.

Außerdem danken möchte ich allen Freunden und Mit-(Ex-)Fachschaftlern, allen voran Jonas, die mich während meiner wunderbaren Zeit in Passau begleitet haben. Ohne euch wären das sicher nicht die bisher interessantesten zehn Jahre meines Lebens geworden. Vielen Dank auch an alle Korrekturleser – die verbliebenen Rechtschreibfehler sind euch gewidmet.

Ganz besonders danken möchte ich meiner Familie, die mir das Studium in Passau erst ermöglicht und mich von Bachelor über Master bis zur Promotion durchgehend unterstützt hat: Danke, dass ihr mir immer Halt und Kraft gebt und ich mich jederzeit und in jeder Situation auf euch verlassen kann. Insbesondere danken möchte ich Andi, der mir gerade in schwierigen Zeiten den Rücken freigehalten hat und immer an meiner Seite war.

Contents

Acknowledgments iii

Contents v

1 Introduction 1
- 1.1 Contribution 2
- 1.2 Outline 4

2 Preliminaries 9

I Constrained Planarity in Theory 15

3 The Hierarchy of Constrained Planarity Problems 17
- 3.1 Problem Definitions 17

4 Planarity 23
- 4.1 PC-trees 24
- 4.2 Planarity of Biconnected Graphs 28

5 Partially Embedded Planarity 33
- 5.1 Partially Embedded Planarity 34
- 5.2 Linear-Time Implementation 51
- 5.3 Conclusion 56

6 Synchronized Planarity 57
- 6.1 Technical Contribution 58
- 6.2 Related Work 60
- 6.3 The Synchronized Planarity Problem 61
- 6.4 Applications 79
- 6.5 Related NP-hard Problems 96
- 6.6 Comparison with the Fulek-Tóth Algorithm 100
- 6.7 Conclusion 101

7 Maintaining Triconnected Components under Node Expansion 103
7.1 Skeleton Decompositions 106
7.2 Extended Skeleton Decompositions 111
7.3 Node Expansion in Extended Skeleton Decompositions 114
7.4 Applications 122
7.5 Conclusion 127

II Constrained Planarity in Practice 129

8 Experimental Comparison of PC-Trees and PQ-Trees 131
8.1 The PC-Tree Update 132
8.2 Our Implementations 134
8.3 Evaluation 149
8.4 Testing Planarity and Generating Embeddings 163
8.5 Conclusion 167

9 Engineering the Synchronized Planarity Algorithm 169
9.1 Related Work 170
9.2 Clustered Planarity in Practice 173
9.3 Engineering Synchronized Planarity 178
9.4 Further Analysis 185
9.5 Conclusion 191

10 Conclusions 195

Bibliography 199

List of Publications 215

1 Introduction

In the digital age, networks are ubiquitous and permeate every facet of modern life. From social interactions to technological infrastructures, networks illustrate the complex interactions of entities and ideas. The mathematical abstraction of a network is known as a graph, a concept that forms the foundation for modeling and analyzing these multifaceted relationships. Graphs provide a versatile language for representing interconnections, whether they depict social networks, biological pathways, computer networks, or abstract relationships between concepts. However, the raw data represented by a graph can be challenging to interpret when not presented visually. This is where the field of graph drawing emerges, a discipline dedicated to making graphs visually comprehensible while faithfully representing their structure.

Studies show that one of the main aspects that negatively affect the readability of a graph visualization are crossings between edges [PAC02; War+02]. In this sense, it would be optimal to have a drawing that is entirely free of such crossings. We call a graph *planar* if it admits an embedding into the plane that has no edge crossings. Planarity is a well-studied concept that facilitates beautiful mathematical structures [BL76; HT73b], allows for more efficient algorithms [Had75], and serves as a cornerstone in the context of network visualization [Pat13].

Yet, in many practical graph drawing applications, we often not only seek an arbitrary drawing that maximizes legibility, but also want to encode additional information via certain aspects of the underlying layout. Examples are *hierarchical* drawings like organizational charts, where we encode a hierarchy among vertices by placing them on predefined levels, *clustered* drawings, where we group vertices by enclosing them in a common region, and *animated* drawings, where dynamic changes to a graph are shown in steps while keeping a static part fixed. Examples for clustered drawings are UML diagrams, where classes are grouped according to the package they are contained in, computer networks, where devices are grouped according to their subnetwork, and integrated circuits, where certain components should be placed close to each other.

It is thus not surprising that various generalizations and extensions of the PLANARITY problem have been studied, see for example [Brü21; DaL15; Sch13]. In the field of constrained planarity we study whether a graph admits a planar drawing that satisfies a given set of constraints. This includes the problems LEVEL PLA-

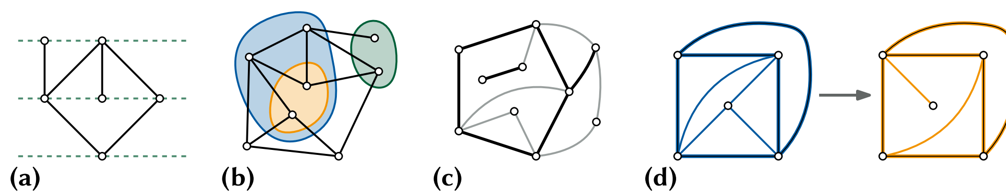


Figure 1.1: Examples of constrained planarity problems: **(a)** Level Planarity, **(b)** Clustered Planarity, **(c)** Partially Embedded Planarity, **(d)** SEFE-2.

narity [Brü21; JLM98] and Clustered Planarity [BR16a; FCE95; Len89], which model the aforementioned hierarchical and clustered drawings, respectively; see Figures 1.1 (a) and 1.1 (b). Animated visualizations of dynamic graphs are for example modeled by Partially Embedded Planarity [Ang+15b; Cha+15; Pat06], where we are already given a planar drawing and want to extend it with new edges, or Simultaneous Embedding with Fixed Edges (SEFE-2) [BKR13; Bra+07; Rut20], where we seek planar drawings of two graphs that coincide on some shared part but have no prescribed embedding; see Figures 1.1 (c) and 1.1 (d). In the last years, the family of constrained planarity problems received a lot of attention in the field of graph drawing. Efficient algorithms were discovered for many of them, while a few others turned out to be NP-complete; see [Sch13] and [DaL15] for an overview. Interestingly, the problems Clustered Planarity and SEFE-2 could not be sorted into either group for a long time, eluding both efficient solutions as well as a proof of NP-hardness. While the former was solved only recently [FT22], the complexity of the latter is still open.

In contrast to the extensive theoretical considerations and the direct motivation by applications, only very few of the found algorithms (many of which have a linear or at most quadratic asymptotic running time) have been implemented and evaluated in practice. This also contrasts the wide variety of implementations available for the different linear-time algorithms for ordinary, i.e., unconstrained planarity [Pat13], which have also been thoroughly assessed in terms of their practical running time [Boy+04a; FMR06]. This lack of implementations for existing constrained planarity algorithms might be in parts due to the high complexity of some of them, for example for (Radial) Level Planarity [Brü21].

1.1 Contribution

The goal of this thesis is to advance the research on both theoretical as well as practical aspects of constrained planarity. On the theoretical side, we consider

two types of constrained planarity problems. The first type are problems that individually constrain the rotations of vertices, that is they restrict the counter-clockwise cyclic orders of the edges incident to vertices. As long as these constraints concern either all or none of the edges incident to a vertex (i.e. they are not partial), the problem is usually solvable in linear time [Ang+10; GKM08; Sch13]. We give a simple linear-time algorithm for the problem Partially Embedded Planarity, which also generalizes to further constrained planarity variants of this type. While a linear-time solution has been known before for this problem [Ang+10], our approach is far simpler and also allows to directly model other planarity variants constraining rotations.

The second type of constrained planarity problem concerns more involved planarity variants that come down to the question whether there are embeddings of one or multiple graphs such that the rotations of certain vertices are in sync in a certain way. Clustered Planarity and the SEFE-2 variant known as Connected SEFE-2, where the subgraph that is shared by all graphs is connected, are well-known problems of this type. Both are generalized by our Synchronized Planarity problem, for which we give a quadratic algorithm. Through reductions from various other problems, we provide a unified modelling framework for almost all known efficiently solvable constrained planarity variants that also directly provides a quadratic-time solution to all of them.

For both algorithms, a key ingredient for reaching a solution is the usage of the right data structure for the problem at hand. Starting on the conceptual level, appropriate data structures are what allows us to model the respective problem in a manageable way in the first place. For example, they give us the tools to describe decompositions into smaller parts as well as describing all choices we have when we seek a planar embedding. Furthermore, the operations these data structures provide for their efficient manipulation are essential for achieving our desired asymptotic running times. Finally, efficient implementations of these data structures also form a cornerstone of practical implementations. In this sense, data structures form the bridge from theoretical to practical solutions, and we will extensively consider both these sides in this thesis. The first important data structure for planar graphs is the SPQR-tree, which succinctly describes all planar embeddings as well as information on the connectivity of the graph. From their inception onward, SPQR-trees form an important tool in handling dynamic changes to graph. We extend the portfolio of dynamic operations on SPQR-trees to the expansion of vertices, which also allows us to further improve the running time of our Synchronized Planarity algorithm. To do so, we give an alternative, axiomatic definition of the SPQR-tree, which makes it easier to reason about its structure.

A further data structure that is even more central to this work is the PC-tree, which describes certain sets of cyclic orders. Analogous to the global description of all planar embeddings of a graph through SPQR-trees, PC-trees can be used to locally describe the possible cyclic orders of edges around vertices in planar embeddings. This makes it a key component for our algorithms, as it allows us to test planarity while also respecting further constraints, and to communicate constraints arising from the surrounding graph structure between vertices with synchronized rotation. Bridging over to the practical side, we resolve several issues with the description of PC-trees that seem to have prevented prior practical realization. This allows us to present the first correct implementation of PC-trees. We also describe further improvements, which allow us to outperform all implementations of alternative data structures (out of which we only found very few to be fully correct) by at least a factor of 4. We show that this yields a simple and competitive planarity test that can also yield an embedding to certify planarity.

We also use our PC-tree implementation to implement our quadratic algorithm for solving Synchronized Planarity. Here, we show that our algorithm greatly outperforms previous attempts at solving related problems like Clustered Planarity in practice. We also engineer its running time and show how degrees of freedom in the theoretical algorithm can be leveraged to yield an up to tenfold speed-up in practice.

Altogether, we add crucial instruments to the algorithmic toolbox for treating constrained planarity problems both in theory and in practice. This is accompanied by improvements to the underlying data structures both on the theoretical as well as the practical side. Based on this, we provide a greatly simplified linear-time solution for various constrained planarity problems that restrict individual vertex rotations. For more involved planarity variants that require the synchronization of rotations between vertices, we describe a quadratic-time solution. This allows us to model a great variety of constrained planarity variants, for which we by means of our implementation also provide a practical solution.

1.2 Outline

The main content of this thesis is divided into a theoretical and a practical part. In the theoretical Part I we introduce two new algorithms for solving constrained planarity problems, show their correctness and analyze their asymptotic running time. In the practical Part II we develop an implementation of a core data structure that both algorithms rely upon and then implement and engineer one of our algorithms based on this. Both parts together are framed by preliminaries in Chapter 2 as well as a conclusion in Chapter 10. In the following, we briefly outline the contents of both parts.

Part I – Constrained Planarity in Theory

This part is concerned with the theoretical description and analysis of solutions to constrained planarity problems. First, we will give an overview over the most important variants of constrained planarity and their relationship in Chapter 3. In the following Chapter 4, we give an exposition of the basic algorithm for standard (i.e., unconstrained) planarity and the PC-tree data structure it relies upon. We will use both in the remainder of this part, where we develop our solutions to the Partially Embedded Planarity and Synchronized Planarity problems.

Partially Embedded Planarity

In the Partially Embedded Planarity problem, we are given a graph G together with a topological drawing of a subgraph H of G, that is, a drawing of H where vertices are represented by points and edges are represented by Jordan curves between their endpoints. We want to test whether the drawing can be extended to a topological drawing of the whole graph such that no two edges cross; see Figure 1.2. Angelini et al. [Ang+10; Ang+15b] gave a linear-time algorithm for solving this problem in 2010. While their paper constitutes a significant result, the algorithm described therein is highly complex: it uses several layers of decompositions according to connectivity of both G and H, its description spans more than 30 pages, and can hardly be considered implementable. In Chapter 5, we give an independent linear-time algorithm that works along the well-known vertex-addition planarity test by Booth and Lueker [BL76; Boo75] (or rather its generalization by Haeupler and Tarjan [HT08]). The key insight here is that the constraints arising from the partial drawing can be formulated as constraints to the rotations of individual vertices. We modify the PC-tree, which is the underlying data structure of the planarity test for representing all planar drawing possibilities, in a natural way to also respect the restrictions given by the prescribed drawing of the subgraph H. The testing algorithm and its proof of correctness only require small adaptations from the comparatively much simpler generic planarity test, of which several implementations exist. If the test succeeds, an embedding can be constructed using the same approaches as for the generic planarity test [Chi+85].

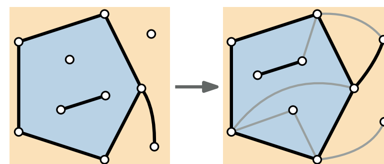


Figure 1.2: The edges added in an instance of Partially Embedded Planarity.

Synchronized Planarity

Many more involved variants of constrained planarity come down to the question whether there are embeddings of one or multiple graphs such that the rotations of certain vertices are in sync in a certain way. This is for example the case in Clustered Planarity, where the order in which edges "leave" a cluster (via its boundary line) needs to line up with the order in which they "enter" the cluster. To formulate this in terms of synchronized vertex rotations, we can split the graph in two at each cluster boundary, in each half contracting the other side of the cluster boundary into a single vertex; see Figure 1.3. Now, the rotations of the two vertices that are obtained from the same cluster boundary need to be in sync.

In Chapter 6, we introduce the problem Synchronized Planarity to model this synchronization. Roughly speaking, its input is a loop-free multi-graph together with synchronization constraints that match pairs of vertices of equal degree by providing a bijection between their edges. Synchronized Planarity then asks whether the graph admits a crossing-free embedding in the plane such that the orders of edges around synchronized vertices are consistent. We show, on the one hand, that Synchronized Planarity can be solved in quadratic time and, on the other hand, that it serves as a powerful modeling language that lets us easily formulate several constrained planarity problems as instances of Synchronized Planarity. In particular, we use this to improve the running time for Clustered Planarity from $O((n+d)^8)$ [FT22] to $O((n+d)^2)$, where n is the number of vertices and d is the total number of crossings between cluster borders and edges. For Connected SEFE-2, we improve the running time from $O(n^{16})$ [FT22] to $O(n^2)$.

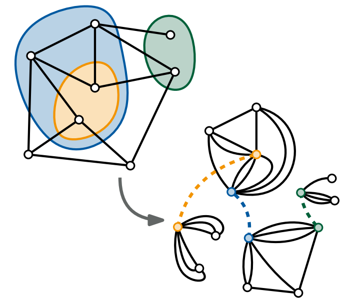


Figure 1.3: Equivalent instances of Clustered- and Synchronized Planarity.

Maintaining Triconnected Components under Node Expansion

The last chapter of Part I is dedicated to a data structure that is, in itself, independent of our algorithmic applications. The SPQR-tree data structure models the decomposition of a biconnected graph into triconnected components, which describe the sets of vertices that cannot be separated by the removal of at most two vertices. From their inception onwards, they have always had a strong relation to dynamic algorithms maintaining information, e.g., on planar embeddings

and connectivity, under edge insertion and, later on, also deletion. In Chapter 7, we study the problem of dynamically maintaining an SPQR-tree while expanding vertices into arbitrary biconnected graphs; see Figure 1.4. This allows us to efficiently merge two SPQR-trees by identifying the edges incident to two vertices with each other. We do this working along an axiomatic definition, thereby lifting the SPQR-tree to a stand-alone data structure that can be modified independently from the graph it might have been derived from. Making changes to this structure, we can now observe how the graph represented by the SPQR-tree changes, instead of having to reason which updates to the SPQR-tree are necessary after a change to the represented graph.

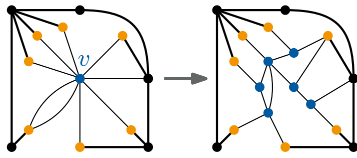


Figure 1.4: Expanding vertex v.

Using efficient expansions and merges of SPQR-trees allows us to further improve the running time of the Synchronized Planarity algorithm from $O(m^2)$ to $O(m \cdot \Delta)$, where m is the number of edges and Δ is the maximum degree of a vertex involved in a synchronization constraint. This also reduces the time for solving several related constrained planarity problems, e.g. for Clustered Planarity from $O((n+d)^2)$ to $O(n+d \cdot \Delta)$, where d is the total number of crossings between cluster borders and edges and Δ more specifically denotes the maximum number of edge crossings on a single cluster border.

Part II – Constrained Planarity in Practice

This part is concerned with the practical implementation of solutions to constrained planarity problems. Its first half focuses on implementing the PC-tree as core component to many constrained planarity algorithms. On this foundation, we describe and engineer an implementation of our quadratic algorithm for the Synchronized Planarity problem in the second half.

Experimental Comparison of PQ- and PC-Trees

PQ-trees and PC-trees are data structures that represent sets of linear and cyclic orders, respectively, subject to constraints that specific subsets of elements have to be consecutive. While equivalent to each other, PC-trees are conceptually much simpler than PQ-trees; updating a PC-tree so that a set of elements becomes consecutive requires only a single operation, whereas PQ-trees use an update procedure that is described in terms of nine transformation templates that have to be recursively matched and applied.

Despite these theoretical advantages, for a long time no practical PC-tree implementation was available. This might be due to the original description by Hsu and McConnell [HM03] in some places only sketching the details of the implementation. In Chapter 8, we describe two alternative implementations of PC-trees. For the first one, we follow the approach by Hsu and McConnell, fill in the necessary, previously missing details and also propose improvements on the original algorithm. For the second one, we use a different technique for efficiently representing the tree using a Union-Find data structure. In an extensive experimental evaluation we compare our implementations to a variety of other implementations of PQ-trees that are available on the web as part of academic and other software libraries. Our results show that both PC-tree implementations beat their closest fully correct competitor, the PQ-tree implementation from the OGDF library [Chi+14; Lei97], by a factor of 2 to 4, showing that PC-trees are not only conceptually simpler but also faster in practice. Moreover, we find the Union-Find-based implementation, while having a slightly worse asymptotic running time in theory, to be twice as fast as the one based on the description by Hsu and McConnell. Finally, we show the positive effects this greatly improved performance has on the planarity testing algorithms that strongly rely on PC-trees.

Engineering the Synchronized Planarity Algorithm

Despite substantial prior theoretical progress on constrained planarity problems, only very few of the found solutions have been put into practice and evaluated experimentally. In Chapter 9, we describe our implementation of the quadratic-time algorithm for solving the problem Synchronized Planarity introduced in Chapter 6. Our experimental evaluation on an existing benchmark set shows that even our baseline implementation outperforms all competitors by at least an order of magnitude. We systematically investigate the degrees of freedom in the implementation of the Synchronized Planarity algorithm for larger instances and propose several modifications that further improve the performance. Altogether, this allows us to solve instances with up to 100 vertices in milliseconds and instances with up to 100 000 vertices within a few minutes.

2 Preliminaries

We expect basic familiarity with fundamental graph-theoretic as well as algorithmic concepts [Cor+22; Die17]. A gentle introduction to many of the following concepts related to planarity can also be found in Chapter 1 of the Handbook of Graph Drawing and Visualization [Pat13]. The purpose of these preliminaries is to establish our terminology, give formal definitions for the notation we use and to introduce some lesser-known concepts used in this thesis.

Sets. A *partition* of a base set X is a grouping of its elements into non-empty subsets, the *cells*, so that every element is contained in exactly one cell. We use $A \mathbin{\dot\cup} B$ to denote the union of two disjoint sets A, B. For a bijection or matching ϕ, we call $\phi(x)$ the *partner* of an element x. When considering running times, we assume a set implementation allowing constant-time insertion and removal of elements, such as doubly-linked lists with pointers stored with the elements.

Linear and Cyclic Orders. Let X be a ground set. Two linear orders α, β of X are *equivalent*, denoted by $\alpha \sim \beta$, if there exist linear orders α_1, α_2 such that $\alpha = \alpha_1\alpha_2$ and $\beta = \alpha_2\alpha_1$. That is, two orders are equivalent if one is a rotation of the other. A *cyclic order* σ is an equivalence class of $\sim$. Given a linear order α, we write $[\alpha] := \{\beta \mid \alpha \sim \beta\}$ for the corresponding cyclic order. For an order σ, we use $\overline{\sigma}$ to denote its reversal.

Let A be a subset of X with $\emptyset \neq A \subseteq X$, let α be a linear order of X and let σ be a cyclic order of X. The set A is *consecutive* in α if $\alpha = \alpha_1\alpha_2\alpha_3$, such that α_2 is a linear order of A and $\alpha_1\alpha_3$ is a linear order of the elements in $X \setminus A$. The set A is *consecutive* in a cyclic order σ if there exists a linear order $\beta \in \sigma$ such that A is consecutive in β. Now let σ be a cyclic order such that A is consecutive in σ and let $a \notin X$. We denote by $\sigma[A]$ the cyclic order of A that is obtained from σ by removing the elements of $X \setminus A$. We denote by $\sigma[A \to a]$ the cyclic order of $(X \setminus A) \cup \{a\}$ obtained from σ by replacing the elements of A with the single element a.

Let σ, τ be cyclic orders of X_1, X_2 with $X_1 \cap X_2 = \{\ell\}$. The *merge* of σ and τ, denoted by $\sigma \otimes_\ell \tau$ is the cyclic order that is obtained by merging the two orders at ℓ. More precisely, let $\sigma = [\alpha\ell]$ and $\tau = [\ell\beta]$, then $\sigma \otimes_\ell \tau = [\alpha\beta]$. Note that this yields $\sigma = (\sigma \otimes_\ell \tau)[X_2 \to \ell]$. If one side only contains the single element ℓ, the merge will effectively remove ℓ from the other side, i.e., $[\alpha\ell] \otimes_\ell [\ell] = [\alpha]$.

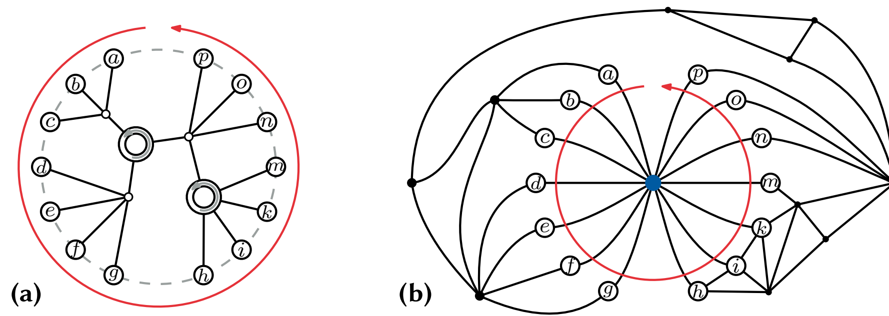


Figure 2.1: (a) A PC-tree T on the set $X = \{a, \ldots, p\}$. Small black disks are P-nodes, larger white disks are C-nodes with their up-to-reversal fixed rotation indicated. The red arrow indicates the currently shown cyclic order, which follows the alphabet. **(b)** A planar drawing of a graph G with a vertex v marked in blue and the edges incident to v identified with X. The rotation of v indicated by the red arrow coincides with the cyclic order from (a). Moreover, T represents all possible rotations of v in any planar drawing of G, that is T is the embedding tree of v.

PQ- and PC-Trees. A *PC-tree* T is a tree without degree-2 vertices whose inner nodes are partitioned into *P-nodes* and *C-nodes*; see Figure 2.1 (a). Edges incident to C-nodes have a cyclic order that is fixed up to reversal, whereas edges incident to P-nodes can be reordered arbitrarily. Traversing the tree according to fixed orders around the inner nodes determines a cyclic order of the leaves X of the tree. Any cyclic order of X that can be obtained from T after arbitrarily reordering the edges around P-nodes and reversing orders around C-nodes is an *admissible order* of X. In this way a PC-tree represents a set of cyclic orders of X; see Figure 2.1. Further examples of PC-trees are given in Figure 4.1 in Section 4.1, where we also discuss this data structure in more detail. The main operation of PC-trees is the update, which modifies the tree to restrict its admissible orders to those where the leaves of a set A called *restriction* are consecutive. This operation is discussed in more detail in Section 4.1 and Chapter 8.

Rooted PC-trees have initially been studied by Booth and Lueker under the name PQ-tree (also referring to C-nodes as Q-nodes) [BL76], where a leaf used as root indicates where the cyclic order is cut into a linear one. There is a linear-time equivalence between the rooted PQ-trees and the unrooted PC-trees [Hsu01]. For theoretical purposes, we thus do not distinguish them and use both terms interchangeably. We will preferably use the term PC-tree and only use the term PQ-tree when it is more suitable for historic reasons, e.g. when used in the name of a known problem. Still, there are several practical differences between both data structures, which we discuss in more detail in Chapter 8.

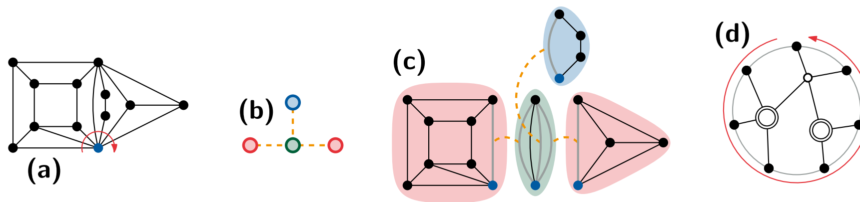


Figure 2.2: A planar graph **(a)**, its SPQR-tree **(b)** and the corresponding skeletons **(c)**. Rigids are highlighted in red, parallels in green, and series in blue. The embedding tree of the vertex marked in blue **(d)**. Small black disks are P-nodes, larger white disks are C-nodes.

Graphs. In the context of this work, when referring to a graph $G = (V, E)$, we usually mean a loop-free multi-graph with vertices V and (possibly parallel) edges E. We set $n = |V|$ and $m = |E|$ and use $n+m$ to describe the size of a graph. For a vertex v, we denote its open neighborhood (excluding v itself) by $N(v)$. A *star* consists of a center vertex connected to its pairwise-nonadjacent *ray vertices*. In a *multi-star*, there may be multiple parallel edges between the center and each ray vertex. A *(k-)wheel* is a (k-)cycle, where each node is also connected to an additional central node. An *st-ordering* is an ordering of the vertices of graph such that the very first and last vertices, usually called s and t, respectively, are adjacent and moreover, each vertex except for s and t has a neighbor which comes earlier in the ordering, and a neighbor which comes later [ET76]. When considering implementations, we assume a graph representation that allows efficient manipulation, such as an adjacency list with doubly-linked lists. This allows inserting vertices and edges in constant time. An edge can be deleted in constant time, while a vertex can be deleted in time linear in its degree.

Connectivity. A separating k-set is a set of k vertices whose removal increases the number of connected components. Separating 1-sets are called *cut-vertices*, while separating 2-sets are called *separation pairs*. A connected graph is *biconnected* if it does not have a cut-vertex. A biconnected graph is *triconnected* if it does not have a separation pair. Maximal biconnected subgraphs are called *blocks*. A cut-vertex may be incident to multiple blocks and a block may be incident to multiple cut-vertices. This adjacency of biconnected components in a connected component yields a bipartite acyclic graph called the block-cut-tree or *BC-tree* for short. A vertex that is not a cut-vertex and thus resides within a unique block is called *block-vertex*.

Each separation pair divides the graph into *bridges*, the maximal subgraphs which cannot be disconnected by splitting the vertices of the separation pair. Splitting

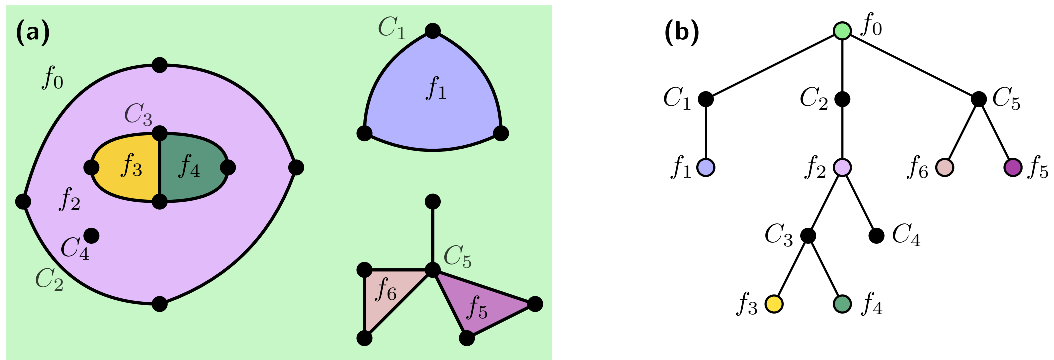


Figure 2.3: Embedding of a disconnected graph **(a)** with its component–face tree **(b)**.

a vertex v introduces a further copy v' of v such that v and v' are non-adjacent and reassigns some of the edges of v to v'. Hopcroft and Tarjan [HT73b] defined a graph decomposition into triconnected components, also called *SPQR-tree* [DT96a], where the components come in three shapes: *bonds* consist of two *pole* vertices connected by multiple parallel edges, *polygons* consist of a simple cycle, and *rigids*, whose embeddings are unique up to reflection; see Figure 2.2. Each edge of these components is either *real*, representing a single edge of the original graph, or *virtual*, representing a subgraph. Each virtual edge has a separation pair as its endpoints and is matched with a virtual edge in another component with equivalent endpoints, yielding a tree structure of the triconnected components. This decomposition can be computed in linear time [GM00]. SPQR-trees are discussed in more detail in Chapter 7.

Planar Drawings and Embeddings. A *(topological) drawing* Γ of a graph $G = (V, E)$ maps each vertex to a distinct point in $\mathbb{R}^2$ and each edge to a Jordan arc in $\mathbb{R}^2$ that connects its two endpoints in such a way that the points of vertices are only contained in arcs for which they are an endpoint. A drawing is *planar* if no two edges share an interior point. The graph G is planar if a planar drawing of G exists. A planar drawing Γ partitions the remainder of $\mathbb{R}^2$ into *faces*; the connected components of $\mathbb{R}^2 \setminus \Gamma$. The single unbounded face is called the *outer face*.

Two planar drawings Γ_1, Γ_2 are *equivalent*, if there exists an ambient isotopy that transforms Γ_1 into Γ_2, i.e., there exists a continuous map $F\colon \mathbb{R}^2 \times [0,1] \to \mathbb{R}^2$ where each of the maps $F_t(x) := F(x,t)$ is a homeomorphism of $\mathbb{R}^2$ such that F_0 is the identity and F_1 maps Γ_1 to Γ_2. An equivalence class of planar drawings is called an *(combinatorial) embedding*. For an embedding $\mathcal{E}$, we use $\mathcal{E}(v)$ to denote the cyclic order of the edges $E(v)$ incident to v as given by $\mathcal{E}$. We also call $\mathcal{E}(v)$

the *rotation* of v; see Figure 2.1 (b). For a connected graph G an embedding $\mathcal{E}$ can be described by a *rotation system*, which describes the rotation $\mathcal{E}(v)$ of each vertex v, together with a designated outer face. Every embedding of a biconnected planar graph can be obtained from an arbitrary embedding by, in the SPQR-tree, flipping its rigids and reordering the parallel edges in its bonds [HT73b]. If G is not connected, we need additional information about the *relative positions* of the connected components. The relative position of a connected component C_1 in relation to another connected component C_2 is the face of C_2 that contains C_1. Note that the set of faces of C_2 depends on the rotation system of C_2. The relative positions of all connected components, given a rotation system for each one, can be encoded by a component–face tree [Ang+15b], an example of which is show in Figure 2.3. The component–face tree consists of one vertex for each connected component and one vertex for each face. Two vertices are adjacent if the respective connected component and face are incident.

An *embedding tree* is a PC-tree that describes all possible rotations of a block-vertex in any embedding [BL76]; see Figures 2.1 (a) and 2.2 (d). Embedding trees are discussed in more detail in Section 7.4.1. A linear-time algorithm that computes an embedding tree as a byproduct of a planarity test is given in Section 4.2.

Part I

Constrained Planarity in Theory

3 The Hierarchy of Constrained Planarity Problems

In this chapter, we introduce several important variants of constrained planarity and discuss their relationships, building on the work of Schaefer [Sch13] and Da Lozzo [DaL15]. Schaefer [Sch13, Figure 2] introduces a hierarchy on the variants of constrained planarity that have been studied in the past. Da Lozzo gives an extended version of this figure, incorporating updates up to 2015 [DaL15, Figure 0.1]. A version of this hierarchy updated to match the current state is given in Figure 3.1. In the figure, arrows indicate that the target problem either generalizes the source problem or solves it via a reduction. See Chapter 6 and especially Section 6.4 for the new relations and the problem Synchronized Planarity marked in blue. In the 2015 version of Da Lozzo, the problems Strip, Clustered and Synchronized Planarity as well as (Connected) SEFE-2 still formed a frontier of problems with unknown complexity, separating efficiently solvable problems from those that are NP-hard. Since then many of these problems were settled in P, especially due to the Clustered Planarity solution from 2019 by Fulek and Tóth [FT22]. The problems Partially PQ-constrained Planarity and strict 1-Fixed Constrained Planarity were initially solved on general graphs through our Synchronized Planarity reduction; see Section 6.4. The only problem from this hierarchy that remains with an unknown complexity is SEFE-2.

To provide a central up-to-date source for information on the various related constrained planarity variants, we made the hierarchy from Figure 3.1 together with definitions of the shown problems and their reductions available online at constrained-planarity.github.io. Following a collaborative model, future updates and modifications to the information presented there can easily be contributed via GitHub.

3.1 Problem Definitions

In this section, we will give definitions and a short background for the most important problems from Figure 3.1. Furthermore, we will highlight our changes to the hierarchy, which are indicated in blue in the figure. See the works by Schaefer [Sch13] and Da Lozzo [DaL15] for full definitions of the remaining problems as well as the reductions between them.

Figure 3.1: The hierarchy of constrained planarity problems. Figure based on the work by Schaefer [Sch13] and Da Lozzo [DaL15]. Changes discussed in this thesis are indicated in blue.

3.1.1 Partially Embedded Planarity

In Partially Embedded Planarity, the undirected graph $G = (V, E)$ is accompanied by a *prescribed* subgraph H for which a planar embedding $\mathcal{H}$ is given. The triplet $(G, H, \mathcal{H})$ is often called *partially embedded graph (PEG)*. We call a planar embedding $\mathcal{G}$ of G an *extension* of the embedding $\mathcal{H}$ of H if the restriction of $\mathcal{G}$ to H coincides with $\mathcal{H}$. The problem Partially Embedded Planarity asks whether such an extension exists for a PEG; see Figures 1.1 (c) and 5.1. Angelini et al. give a linear-time algorithm for testing this [Ang+15b]. We give an independent, simpler linear-time algorithm in Chapter 5. Schaefer shows that Partially Embedded Planarity can be reduced to SEFE-2, while we give a reduction to the Clustered Planarity problem in Section 6.4.3, which lies further up in the hierarchy of Figure 3.1.

3.1.2 Level Planarity

In Level Planarity, the directed graph $G = (V, E)$ is equipped with a function $\gamma : V \longrightarrow \{1, 2, \dots, k\}$ with $k \in \mathbb{N}$ such that for every edge $(u, v) \in E$ it is $\gamma(u) < \gamma(v)$. The level graph is called *proper* if it is $\gamma(u)+1 = \gamma(v)$ for every edge. With $V_i = \gamma^{-1}(i)$ we denote all vertices on level i. A *level-planar drawing* maps all vertices $v \in V_i$ of a level i to a point on the line $y = i$ and represents each edge as y-monotone curve such that no two edges cross; see Figure 1.1 (a). A level graph is level planar if it admits a level-planar drawing. Jünger et al. give a linear-time algorithm for testing level planarity [JLM98].

Radial Level Planarity is a generalization of Level Planarity, where levels are not represented by axis-parallel lines, but by concentric circles. Equivalently, this can be seen as drawing the horizontal levels on a standing cylinder instead of in the plane. Testing level planarity can be reduced to the radial variant by introducing an edge from the lowest to the highest level, at which the cylinder can be cut to transform a radial solution back into the plane [Sch13]. Bachmaier et al. give a linear-time algorithm for testing radial level planarity [BBF05]. Schaefer shows that Level Planarity can be reduced to Radial Level Planarity, which can in turn be reduced to Clustered Planarity [Sch13]. We give an alternative reduction using fewer clusters in Section 6.4.4.

Strip Planarity is a variant of Level Planarity where levels are not represented by horizontal lines but horizontal strips, which allow their contained vertices to be slightly shifted vertically. Angelini et al. [Ang+16; DaL15] remark that it coincides with the case of the Atomic Embeddability problem where the host graph is a path. See Sections 6.2 and 6.4.1 for more details on this problem

and our quadratic-time solution to it. Angelini et al. give a cubic-time algorithm for Strip Planarity if the combinatorial embedding of the underlying graph is fixed [Ang+16].

3.1.3 Partially (F)PQ-constrained Planarity

In Partially PQ-constrained Planarity, we are given a graph $G = (V, E)$ together with a PQ-tree $T(v)$ for each of its vertices $v \in V$, where the leaves $L(T(v))$ are a (not necessarily strict) subset of the edges $E(v)$ incident to v. We seek an embedding $\mathcal{E}$ where, for each vertex v, the order of incident edges $\mathcal{E}(v)$ is admissible by its PQ-tree $T(v)$; see Figure 6.11. Gutwenger et al. [GKM08] give a linear-time algorithm for a non-partial variant of the problem on general graphs, that is where all incident edges need to be leaves of the PQ-tree, under the name EC-Planarity. They also extend the restrictions to so-called *FPQ-trees*, which also contain *F-nodes* that are similar to Q-nodes but have an entirely fixed rotation without allowing flips. Schaefer [Sch13] discusses even further restricted variants where for each vertex, all incident edges are restricted by a single P- or F-node (Partial Rotation) or by a single P- or Q-nodes (Partial Rotation with flips). Bläsius and Rutter give a linear-time algorithm for Partially PQ-constrained Planarity on biconnected graphs [BR16b]. They also note that their algorithm can easily be extended to the case of FPQ-trees. Note that this makes the problem equivalent to EC-Planarity with Free Edges as presented by Gutwenger et al. [GKM08]. We give a solution that can also handle this case in quadratic time on general graphs in Section 6.4.7.

3.1.4 Clustered Planarity

In Clustered Planarity, the embedding has to respect a laminar family of clusters, that is every vertex is part of some (hierarchically nested) cluster and an edge may only cross a cluster boundary if it connects a vertex from the inside of the cluster with one from the outside [BR16a; Len89]; see Figures 1.1 (b) and 6.6. Formally, we equip the undirected graph $G = (V, E)$ with a cluster hierarchy T, which is a rooted tree with the vertices V as leaves. Each inner node μ of T represents a *cluster* encompassing all leaves V_μ of the subtree rooted at μ. A clustered-planar drawing is a planar drawing of G that also maps every cluster μ to a simply connected closed region R_μ such that

1. R_μ encloses exactly the vertices in V_μ,
2. no two cluster region boundaries intersect, and
3. no edge intersects the boundary of a cluster more than once.

A cluster graph is clustered-planar if it admits a clustered-planar drawing.

Lengauer [Len89] studied and solved this problem as early as 1989 in the setting where the clusters are connected. Feng et al. [FCE95], who coined the term Clustered Planarity, rediscovered this algorithm and asked the general question where disconnected clusters are allowed. This question remained open for 30 years. In that time, polynomial-time algorithms were found for many special-cases [AD19; Cor+08; Ful+15; Gut+02], see [BR16a] for an overview, before Fulek and Tóth [FT22] found an $O((n+d)^8)$ solution in 2019, where d is the number of crossings between a cluster-border and an edge leaving the cluster. We give more details on this in Sections 6.2 and 6.6.

3.1.5 Partitioned 2-page Book Embedding

In the Partitioned $\mathcal{T}$-coherent 2-page Book Embedding problem, we are given a PC-tree T consisting of only P-nodes and leaves V together with two sets $E_1, E_2 \subseteq \binom{V}{2}$. We seek an order σ of V such that σ is admissible by T and for no pair of edges from the same set E_i (with $i \in \{1,2\}$) the endpoints alternate in σ [Ang+12]. Conceptionally, the problem asks whether the vertices V can be placed along the spine of a book such that their order is admissible by T and that the edges of E_1 and E_2 can be drawn planarly on two separate pages. The case where T is trivial (i.e., consists of a single inner P-node) is called Partitioned 2-page Book Embedding and can be solved in linear time [ABD12; HN09; HN18].

3.1.6 Simultaneous Embedding with Fixed Edges (SEFE-2)

In SEFE-2, we are given two graphs that share some vertices and edges and we want to embed both graphs individually such that their common parts are embedded the same way [BKR13; Bra+07; Rut20]; see Figures 1.1 (d) and 6.10. More general variants of SEFE are often NP-complete, e.g., SEFE-3 with three given graphs [Gas+06], even if all share the same common part [ADN15; Sch13]. In contrast, more restricted variants are often efficiently solvable, e.g., when the shared graph is biconnected, a star, a set of cycles, or has a fixed embedding [Ang+12; Ang+15b; BR15]. The case where the shared graph is connected, which is called Connected SEFE-2, was shown to be equivalent to the Partitioned $\mathcal{T}$-coherent 2-page Book Embedding problem [Ang+12] and to be reducable to Clustered Planarity [AD16], all of which were recently shown to be efficiently solvable [FT22]. We give a quadratic solution in Section 6.4.6. In contrast to these results, the complexity of the general SEFE-2 problem with two graphs sharing an arbitrary common graph is still unknown.

4 Planarity

This chapter is based on joint work with Ignaz Rutter and Sandhya T. P. which is currently under review [8].

In this chapter, we will give an overview over the core concepts underlying the vertex-addition planarity test by Booth and Lueker [BL76; Boo75]. Our exposition follows the more general approach by Haeupler and Tarjan [HT08]. The test incrementally inserts the vertices of the graph in an order that ensures that the not-yet inserted vertices form a connected subgraph. This allows us to assume that at every step, by the Jordan Arc Theorem, all edges to not inserted vertices have to lie on the outer face of the already inserted subgraph. We will only care about the possible cyclic orders of such edges on the outer face, disregarding the different planar embeddings that yield these orders. To insert the next vertex, all edges connecting it to already inserted vertices need to be consecutive on the outer face, as otherwise further edges to not-yet inserted vertices would be enclosed. We use PC-trees to efficiently represent the set of possible edge orders on the outer face and to restrict it to only those orders that have a certain subset of edges consecutive. We describe this data structure in the following Section 4.1 in more detail, while explaining the details of the planarity test based on it in Section 4.2 following thereafter.

Note that the initial idea for this planarity test was given by Lempel, Even and Cederbaum [LEC67], albeit with a quadratic running time. A linear-time implementation was made possible by the PQ-trees described by Booth and Lueker [BL76; Boo75], together with a linear-time algorithm for finding the required st-ordering given a decomposition into biconnected components [ET76]. This planarity test was later generalized by Haeupler and Tarjan [HT08] to directly work on non-biconnected graphs using the PC-trees devised by Hsu and McConnell [HM03; HM04]. For the sake of simplicity, we will describe the Haeupler and Tarjan test using PC-trees only on biconnected graphs here, as it can easily be generalized to non-biconnected graphs using a decomposition into their biconnected components. We will later show how the planarity test can be adapted to also directly test non-biconnected graphs in Section 5.1.4. We refer to the book chapter by Patrignani [Pat13] for a more detailed overview over the history of planarity tests.

A comparison how the test and PQ-trees by Booth and Lueker differ from the concepts we present here can also be found there as well as in Chapter 8, although these differences are insignificant for this algorithm.

4.1 PC-trees

Recall that PC-trees represent cyclic orders of a base set X, where a PC-tree T is an unrooted tree with leaves X and inner nodes of degree at least 3, each of which is either a *P-node* or a *C-node*. Examples of PC-trees are given in Figure 4.1, where P-nodes are denoted by small circles whereas C-nodes are denoted by larger double circles. While the edges incident to a P-node can be rearranged without any restriction, the edges incident to a C-node come with a cyclic order that is fixed up to reversal. Any embedding of a PC-tree T that respects this constraint induces a cyclic order of its leaves that we call an *admissible order* of T. The set of all admissible orders of T is denoted by $\omega(T)$. We define the special *null-tree* to be the PC-tree with $\omega(T) = \emptyset$. We refer to the set of all leaves of T as $L(T) = X$. Note that a P-node with three neighbors allows the same permutations as a C-node of the same degree. We thus assume P-nodes to have degree at least 4. We consider a PC-tree *trivial* if it consists of a single inner P-node (with at least four leaves). Otherwise, it consists of a single C-node with at least two leaves, or it contains at least two inner nodes, all of which have degree at least 3.

Let $\emptyset \neq A \subsetneq L(T)$. We now want to classify when A is consecutive in every admissible order of T, that is whether T allows for any cyclic order that has A non-consecutive. An edge e of T is *consistent* with A if one of the two subtrees obtained by removing e contains only leaves from A. We denote by $A(e) \subseteq A$ the leaves of this subtree. For two consistent edges e, e' of T, we define $e \prec e'$ if $A(e) \subseteq A(e')$. This gives a partial order on the consistent edges of A. We denote by $E(T, A)$ the set that contains all maximal elements of this partial order; see Figure 4.1 (b). We call A *consecutive* (with respect to T) if $E(T, A)$ either consists of a single edge or is a consecutive set of edges around a C-node. Observe that A is consecutive if an only if A is consecutive in every order $\sigma \in \omega(T)$.

In our applications, we need the following basic operations of PC-trees: `Merge`, `Split`, `Update`, and `Intersect`; see also Figure 4.1. We now describe each of these in more detail.

`Merge` Let T_1, T_2 be two PC-trees whose respective leaf sets have size at least 2 and that share exactly one leaf ℓ; see Figure 4.1 (c). The `Merge` of T_1, T_2, denoted as $T_1 \otimes_\ell T_2$, is the PC-tree T obtained by identifying the two copies of ℓ in T_1 and T_2

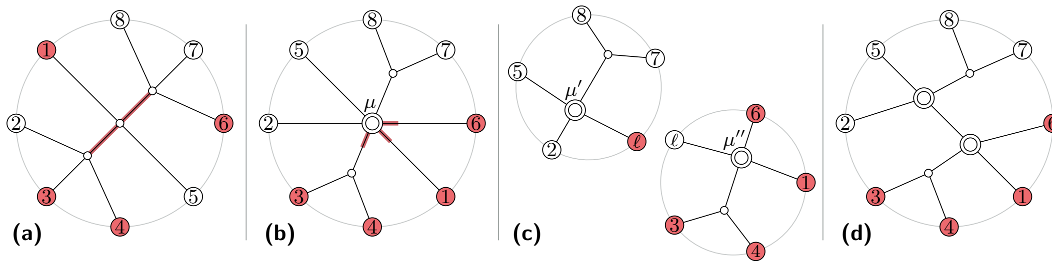
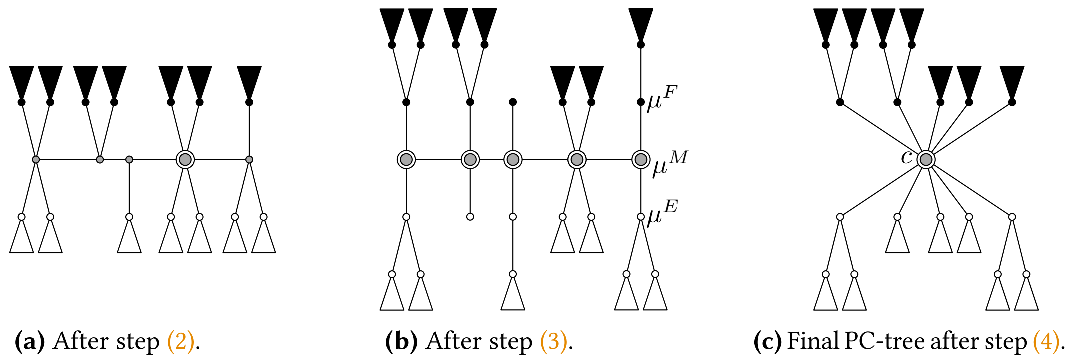


Figure 4.1: **(a)** A PC-tree T on the set $L(T) = \{1, \ldots, 8\}$ with only P-nodes as inner nodes. The red leaves belong to a set A and their terminal path is highlighted in red. **(b)** The PC-tree $T + A$ ensuring that the edges in A are consecutive. Here, $E(T, A)$ consists of the three edges marked in red incident to C-node μ. **(c)** The PC-trees $T' = (T + A)[A \to \ell]$ (left) and $T'' = (T + A)[A^c \to \ell]$ (right) showing the split of the previous tree and how μ is split into two parts μ' and μ'' of the respective trees. **(d)** The PC-tree $T' \otimes_\ell T''$ showing the merge of the previous two trees.

and smoothing the resulting degree-2 node ℓ into an edge xy, where x, y are the two neighbors of ℓ in T_1 and T_2, respectively; see Figure 4.1 (d). Formally, we have $\omega(T) = \{\sigma_1 \otimes_\ell \sigma_2 \mid \sigma_i \in \omega(T_i) \text{ for } i \in \{1, 2\}$. The orders $\sigma_1 \in \omega(T_1)$ and $\sigma_2 \in \omega(T_2)$ corresponding to a $\sigma \in \omega(T)$ can be obtained by undoing the merge that created T from T_1 and T_2 while maintaining the embedding of T that corresponds to σ. Observe that the leaves of each input tree are consecutive in the tree resulting from the merge, i.e., $L(T_i)$ is consecutive with respect to T for $i = 1, 2$. Furthermore, any PC-tree can be obtained by merging trees with a single inner node.

We also extend the definition of Merge to the case where one tree, say T_2, consists only of a single leaf ℓ. In this case, analogously to cyclic orders, we simply remove ℓ from T_1.

Split Let T be a PC-tree and let set A with $\emptyset \neq A \subsetneq L(T)$ be consecutive with respect to T. The operation Split separates T into two new PC-trees T' and T'' representing the admissible orders of $A^c = L(T) \setminus A$ and A, respectively, in T; see also Figure 4.1 (c). The two trees have leaves $L(T') = A^c \cup \{a\}$ and $L(T'') = A \cup \{a\}$, where $a \notin L(T)$ is a new leaf that represents the position of the split-off subtree in each of the resulting halves. The PC-tree T' is obtained by replacing the edges in $E(T, A)$ by the single new leaf a and removing the subtrees containing the leaves in A. Symmetrically, the PC-tree T'' is obtained by replacing the edges in $E(T, A^c)$ with a and removing the subtrees



(a) After step (2). **(b)** After step (3). **(c)** Final PC-tree after step (4).

Figure 4.2: Visualization of the updates to the terminal path made to ensure a set of leaves is consecutive. The full subtrees with only leaves that should be made consecutive are shown in black, empty subtrees are shown in white. The terminal path is the horizontal line with gray nodes.

containing the leaves in A^c. This yields trees T', T'' with $\omega(T') = \{\sigma[A \to a] \mid \sigma \in \omega(T)\}$ and $\omega(T'') = \{\sigma[A^c \to a] \mid \sigma \in \omega(T)\}$. To refer to one of the resulting trees, we will borrow this notation and write $T[A \to a]$ and $T[A^c \to a]$ for the trees T' and T'', respectively.

`Update` Let T be a PC-tree and let $A \subseteq L(T)$ be a set of leaves. The operation `Update`, denoted as $T+A$, produces a new PC-tree T' with $\omega(T') = \{\sigma \in \omega(T) \mid A$ is consecutive in $\sigma\}$. We also call the set A a *restriction* (of the admissible orders of T to those where A is consecutive). The procedure has the property that the leaf set A is consecutive with respect to the resulting tree T'. We call a restriction *impossible* if there is no admissible order of L where the leaves in R are consecutive, i.e., $T + A$ is the null-tree. Note that leaf sets with $|A| \in \{0, 1, |L(T)| - 1, |L(T)|\}$ are always consecutive and thus do not require changes to T. Otherwise, the required changes are made by the following steps initially described by Hsu and McConnell [HM03; HM04].

(1) Determine the edges of T that are consistent with neither A nor $L(T) \setminus A$; see Figures 4.1 (a) and 4.2. If these edges do not form a path, the so-called *terminal path*, then T does not represent any cyclic order where A is consecutive, and we return the null-tree.

(2) Reorder the edges around the nodes on the terminal path so that all subtrees that have all their leaves in A, which we will call *full*, lie on one side and all subtrees with leaves in $L(T) \setminus A$, which we will call *empty*, lie on the other side; see Figure 4.2 (a). If this is not possible, then this

is due to a C-node around which edges consistent with A and edges consistent with $L(T) \setminus A$ alternate. It follows that T does not represent a cyclic order where A is consecutive, and we return the null-tree.

(3) Split each P-node μ on the terminal path twice, once to move all edges to full subtrees adjacent to μ to a new P-node μ^F and a second time to move all edges to empty subtrees to a new P-node μ^E. Add edges to the new nodes, making the remainder of μ adjacent to μ^F, μ^E, and up to two edges of the terminal path; see Figure 4.2 (b). Convert this remaining part into a C-node μ^M and choose its embedding such that the two terminal edges are not adjacent to each other if it has degree 4, and flip it so that all full subtrees again lie on the same side of the terminal path.

(4) Contract all nodes of the terminal path (which are now all C-nodes) into a single, central C-node. Finally, smooth degree-2 vertices and remove degree-1 vertices that are not leaves of the original tree T; see Figure 4.2 (c).

Hsu and McConnell [HM03; HM04] show that each of these steps can be implemented to run in time proportional to $|A|$ plus the length of the terminal path. This leads to amortized linear time in the size of the set A as all terminal edges disappear through the update. We will describe how this procedure can be implemented efficiently in greater detail in Chapter 8.

`Intersect` Let T_1, T_2 be two PC-trees with the same set of leaves. Then the operation `Intersect` produces a new PC-tree T with $\omega(T) = \omega(T_1) \cap \omega(T_2)$. Booth [Boo75] describes a linear-time algorithm for computing the intersection of two PQ-trees; the same algorithm can be applied for PC-trees [Pfr20]. The general idea is to convert T_1 into a set of consecutivity constraints and to then update T_2 with these sets so that it represents only the orders that are represented by both trees. The key to achieve linear running time is to contract maximal subtrees that are already consecutive in both trees into single nodes.

Note that, unlike for cyclic orders, where splitting and merging are converse operations, the same does not always hold for PC-trees. Figure 4.1 (d) shows an example where a merge following a split does not yield the initial PC-tree. In this example, only certain orders $\sigma_1 \in \omega(T')$ and $\sigma_2 \in \omega(T'')$ can be merged to an admissible order of the original T. We call such pair, that is where $\sigma_1 \otimes_\ell \sigma_2 \in \omega(T)$ holds, *compatible*. The following lemma shows in which case split and merge are converse operations on PC-trees and when orders for a split tree are compatible.

▶ **Lemma 4.1.** Let T be a PC-tree with a consecutive set A, let $T' = T[A \rightarrow \ell]$ and $T'' = T[A^c \rightarrow \ell]$ and let $\sigma_1 \in \omega(T')$ and $\sigma_2 \in \omega(T'')$. If $E(T, A)$ is a single edge e, then $T = T' \otimes_\ell T''$ and $\sigma_1 \otimes_\ell \sigma_2 \in \omega(T)$, that is any pair of σ_1 and σ_2 is compatible. Otherwise, $E(T, A)$ is a set of edges consecutive around a C-node μ of T, and we have $T \neq T' \otimes_\ell T''$ and $\omega(T) \subsetneq \omega(T' \otimes_\ell T'')$. In this case, σ_1 and σ_2 are compatible if and only if they induce the same flip of the split halves of μ in T' and T''. If σ_1 is not compatible with σ_2, it is instead compatible with $\overline{\sigma_2}$. ◀

Proof. If $E(T, A)$ is a single edge e, tree T is split by splitting e. It can thus be reobtained by merging at e again, that is $T = T' \otimes_\ell T''$. As the embeddings that σ_1 and σ_2 induce on T' and T'', respectively, can also be joined at e to obtain an embedding of T, we always have $\sigma_1 \otimes_\ell \sigma_2 \in \omega(T)$.

Otherwise, $E(T, A)$ is a set of edges consecutive around a C-node μ of T. Splitting T into two trees T', T'' also splits μ into two respective C-nodes μ', μ'', where μ'' is incident to the edges in $E(T, A)$ plus ℓ and μ' gets the remaining edges of μ plus another copy of ℓ; see Figure 4.1 (d). Merging T' and T'' now does not yield T again, as μ' and μ'' are still separate C-nodes connected by the edge ℓ in $T^* = T' \otimes_\ell T''$. We have $\omega(T^*) \supsetneq \omega(T)$ as μ' and μ'' can be flipped independently and thus in total allow four different orders for their incident edges in T^*, while μ in T only allows two. The embeddings σ_1 and σ_2 induce on T' and T'' can thus only be merged to an embedding of T if the rotations of μ' and μ'' they induce can be merged to form an admissible rotation of μ. As the C-nodes have two admissible embeddings, either σ_1 and σ_2 are compatible or σ_1 and $\overline{\sigma_2}$ are compatible. ■

4.2 Planarity of Biconnected Graphs

To test planarity for a graph $G = (V, E)$, we will iteratively insert the vertices of the graph in a certain order, that is in each step $i \in \{1, \dots, n\}$ we grow the set $V_i = \{v_1, \dots, v_i\} \subseteq V$ of already-inserted vertices. At each step, we partition the edges of G into three types: *Embedded edges* have both endpoints in V_i, *half-embedded* have exactly one endpoint in V_i and *unembedded edges* have both endpoints in $V \setminus V_i$. When inserting vertex v_i into the graph, its incident unembedded edges become half-embedded and its incident half-embedded edges become embedded. We denote by G_i the subgraph of G induced by V_i, and by G_i^+ the graph that is obtained from G_i by adding each half-embedded edge $e = uv$ with $u \in V_i$ as half-edge that is only incident to u. If the context is clear, we refer to half-embedded edges simply as half-edges.

The central idea of the planarity test is to use a vertex order that has $G[V \setminus V_i]$ connected at each step $i \in \{1, \dots, n\}$. By the Jordan curve theorem, this ensures that all half-embedded edges must be embedded in the same face of G_i^+, without loss of generality, the outer face. We will for now assume G to be biconnected and use an st-ordering $v_1, \dots, v_n$ of its vertices, as this ensures that both $G[V_i]$ and $G[V \setminus V_i]$ are connected. Observe that a planar embedding of G determines a planar embedding of each G_i^+. Let $\Omega(G_i^+)$ denote the set of all embeddings of G_i^+ with all half-edges on the outer face. For an embedding $\mathcal{E} \in \Omega(G_i^+)$ of a connected graph G_i^+, let $\omega(\mathcal{E})$ be its cyclic order of half-edges on the outer face. We define $\omega(G_i^+) = \{\omega(\mathcal{E}) \mid \mathcal{E} \in \Omega(G_i^+)\}$ to be the set of all such orders.

To test the planarity of a biconnected graph G given an st-ordering $v_1, \dots, v_n$ of its vertices, we compute PC-trees $T_1, \dots, T_n$ satisfying the invariant $\omega(G_i^+) = \omega(T_i)$ for all $i \in \{1, \dots, n\}$. The tree T_1 consists of a single P-node with leaves $E(v_1)$. Given a PC-tree T_i, the next PC-tree T_{i+1} is obtained as follows. Conceptually, we make the half-edges F that lead from G_i^+ to v_{i+1} consecutive in T_i and replace them by a single edge leading to a new P-node with leaves $E(v_{i+1}) \setminus F$. Formally, we first turn v_{i+1} into a PC-tree S consisting of a single P-node with leaves $E(v_{i+1})$. We make the edges F between G_i^+ and v_{i+1} consecutive in both T_i and S using the `Update` operation. We split the resulting PC-tree $T_i + F$ into trees $T^F = (T_i + F)[F^c \to \ell]$ and $T' = (T_i + F)[F \to \ell]$, where T^F describes the order of half-edges F leading from G_i^+ to v_{i+1} and T' describes the order of the remaining half-edges of G_i^+. Similarly, we split $S + F$ into $S^F = (S + F)[F^c \to \ell]$ and $S' = (S + F)[F \to \ell]$, where S^F describes the order of half-edges F leading from v_{i+1} to G_i^+ and S' describes the order of the remaining half-edges of v_{i+1}. Note that $L(S^F) = L(T^F) = F \cup \{\ell\}$ and $L(S') \cap L(T') = \{\ell\}$. Furthermore, $L(S') \cup L(T')$ contains all half-embedded edges that are present after step $i + 1$ plus ℓ, that is $L(S') \cup L(T') = L(T_{i+1}) \cup \{\ell\}$. Finally, we merge trees S' and T' at ℓ to obtain T_{i+1} with $\omega(T_{i+1}) = \omega(T' \otimes_\ell S')$.

A full proof of correctness of this approach is folklore, albeit not being explicitly given in the extended abstract by Haeupler and Tarjan [HT08]. As our proof of Lemma 5.2 includes such a proof (in the context of partially embedded planarity), we will only sketch its outline here.

▸ **Lemma 4.2.** For every step $i \in \{1, \dots, n\}$ of the algorithm, it holds that $\omega(G_i^+) = \omega(T_i)$. ◂

Proof sketch. The proof works by induction on the number of steps, where the statement trivially holds for the first step. Assuming that $\omega(G_i^+) = \omega(T_i)$ holds for step i, the statement for the next step $i + 1$ can be shown by arguing both inclusions separately.

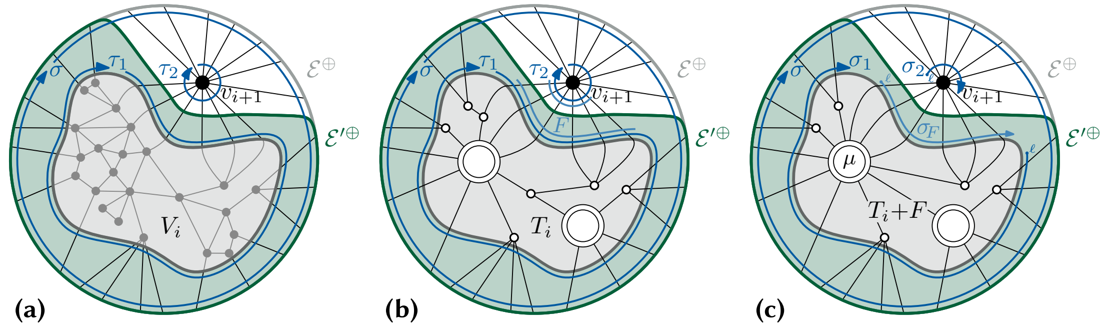


Figure 4.3: (a) A drawing $\mathcal{E}^{\oplus}$ of G^+_{i+1} and the drawing $\mathcal{E}'^{\oplus}$ of G^+_i it contains. **(b)** The PC-tree T_i representing all planar embeddings of V_i. Relevant orders of half-edges are marked in blue. **(c)** The PC-tree $T_i + F$ with a C-node μ.

To show $\omega(G^+_{i+1}) \subseteq \omega(T_{i+1})$, take an order $\sigma \in \omega(G^+_{i+1})$ and let $\mathcal{E} \in \Omega(G^+_{i+1})$ be a corresponding embedding with $\omega(\mathcal{E}) = \sigma$. Let $\mathcal{E}'$ be the embedding of G^+_i obtained by deleting v_{i+1}; see Figure 4.3 (a). It can easily be shown that $\mathcal{E}' \in \Omega(G^+_i)$ and we have, thanks to G_i being connected, $\tau_1 = \omega(\mathcal{E}') \in \omega(G^+_i)$ and, by the inductive hypothesis, $\tau_1 \in \omega(T_i)$. All edges in F must be consecutive in τ_1 and we thus have $\tau_1 \in \omega(T_i + F)$. Using the rotation τ_2 of v_{i+1} in $\mathcal{E}$ we can then show $\sigma = \tau_1[F \to \ell] \otimes_\ell \tau_2[F \to \ell] \in \omega((T_i + F)[F \to \ell] \otimes_\ell (S + F)[F \to \ell]) = \omega(T_{i+1})$; see Figure 4.3 (b).

To conversely show $\omega(G^+_{i+1}) \supseteq \omega(T_{i+1})$, take an order $\sigma \in \omega(T_{i+1})$ and let $\sigma_1 \in T'$ and $\sigma_2 \in S'$ be compatible orders such that $\sigma = \sigma_1 \otimes_\ell \sigma_2$; see Figure 4.3 (c). We can find an order $\sigma_F \in \omega(T^F) \cap \overline{\omega(S^F)}$ of $F \cup \{\ell\}$ such that $\tau_1 = \sigma_1 \otimes_\ell \sigma_F$ not only lies (by construction) in $\omega(T' \otimes_\ell T^F)$, but also in $\omega(T_i + F) \subseteq \omega(T_i)$ due to Lemma 4.1. By the inductive hypothesis, we have $\tau_1 \in \omega(G^+_i)$ and there exists an embedding $\mathcal{E}' \in \Omega(G^+_i)$ with $\omega(\mathcal{E}') = \tau_1$. Furthermore, we always have $\tau_2 = \sigma_2 \otimes_\ell \overline{\sigma_F} \in \omega(S+F) \subseteq \omega(S)$ as $E(S+F, F)$ is a single edge. This is because in S, the leaves F are all adjacent to the same P-node (which is the only inner node of S). We choose τ_2 as rotation for v_{i+1} and add it to $\mathcal{E}'$ to obtain a planar embedding $\mathcal{E}$ of G^+_{i+1}. We can show that this results in $\omega(\mathcal{E}) = \tau_1[F \to \ell] \otimes_\ell \tau_2[F \to \ell] = \sigma_1 \otimes_\ell \sigma_2 = \sigma \in \omega(G^+_{i+1})$. ■

Note that if none of the steps fails due to an impossible update, this means that we found (implicit) planar embeddings for all considered subgraphs. Finding a non-null PC-tree T_{n-1} then suffices to show planarity, as in the last step we would make all leaves of this tree consecutive (which is always possible) and replace them with v_n, which has no further half-edges to not-yet inserted vertices. We note that T_{n-1} describes all planar embeddings of $G[V \setminus \{v_n\}]$, that is, it is the embedding tree of v_n. Conversely, if the process fails at any step, this is due to a subgraph having no

Algorithm 1: Test a biconnected graph G for planarity.

```
1 v_1, ..., v_n ← st-Order(G);
2 T_1 ← single P-node with leaves E(v_1);
3 for i in 1, ..., n − 2 do
4     S ← single P-node with leaves E(v_{i+1});
5     F ← edges between G_i and v_{i+1} in G;
6     S′ ← (S + F)[F → ℓ];
7     T′ ← (T_i + F)[F → ℓ];
8     T_{i+1} ← S′ ⊗_ℓ T′;
9 return true if no Update returned the null tree, and false otherwise;
```

planar embedding in which all edges to the next vertex are consecutive, meaning that the graph is non-planar. Algorithm 1 illustrates this algorithm in pseudo code. Hsu and McConnell show (as Booth and Lueker already did for PQ-trees [BL76; Boo75]) that the PC-tree updates can be done in amortized linear time [HM03; HM04]. As a decomposition into biconnected components as well as corresponding st-orderings can be found in linear time [ET76; HT73a], the overall planarity test thus runs in linear time [BL76; HT08].

Chiba et al. [Chi+85] show that an embedding can be constructed by storing for each step one admissible order of $\omega(T^F) \cap \overline{\omega(S^F)}$ for the edges in F, although special care must taken as some of these orders may be reversed in later steps. This is the case when one order depends on the flip of a split C-node μ^F, whose other half μ' (which remains part of the PC-tree in the next step) gets flipped in a later step. Still, this issue can be resolved by keeping track of such flips and we can generate an embedding along the planarity test in linear time [Chi+85]. To summarize, we have the following theorem.

▶ **Theorem 4.3 (Booth and Lueker [BL76], Chiba et al. [Chi+85]).** Planarity of a biconnected graph can be tested in linear time. A corresponding planar embedding can be constructed at the same time. This extends to general graphs using a decomposition into biconnected components. ◀

5 Partially Embedded Planarity

This chapter is based on joint work with Ignaz Rutter and Sandhya T. P. which is currently under review [8].

In the *partial representation extension problem*, the input consists of a graph G, and a representation $\mathcal{H}$ of a subgraph $H \subseteq G$. The question is whether there exists a representation $\mathcal{G}$ of G whose restriction to H coincides with $\mathcal{H}$. The complexity of the problem strongly varies with the type of representation that is considered. For planar straight-line drawings, the problem was shown to be NP-hard [Pat06], and in fact recently turned out to be $\exists\mathbb{R}$-complete [LMM18]. For various other classes of representations, a plethora of algorithmic and complexity results have been established in recent years. For example, Klavík et al., who coined the term *partial representation extension*, solved the problem for interval representations [KKV11] in quadratic time, which they later improved to linear [Kla+16]. Shortly afterwards, Angelini et al. [Ang+10; Ang+15b] gave a linear-time algorithm for extending planar topological drawings. Since then, the problem has been studied for a variety of different types of intersection representations, e.g., proper and unit interval graphs [Kla+17], permutation graphs [Kla+12], circle graphs [CFK19], contact representations of geometric objects [Cha+14], trapezoid graphs [KW17] and rectangular duals [Cha+21]. In the context of drawings, the problem has also been studied for orthogonal drawings [ART21].

We focus on the case of planar topological drawings, for which Angelini et al. [Ang+15b] gave a linear-time algorithm. Their paper first gives a combinatorial characterization for yes-instances of Partially Embedded Planarity. The authors show that it is necessary and sufficient for a yes-instance to respect both the cyclic edge orders around vertices and the relative positions of different connected components defined by $\mathcal{H}$. In particular, for a biconnected graph G, these "compatibility constraints" set out by $\mathcal{H}$ can be individually verified on the nodes of the SPQR-tree of G [DT96b; Pat13]. This procedure can also be used to individually test each block of a connected graph. However, it is also necessary to verify certain compatibility constraints between the different blocks. Similarly, the authors show that in the disconnected case, Partially Embedded Planarity can be solved by testing each connected component and verifying the compatibility of the relative positions for the different components. This characterization leads to a polynomial-

time algorithm by progressively decomposing the input graph into its connected, biconnected, and triconnected components, while also verifying the compatibility constraints. In order to improve the running time to linear, the authors first give a linear-time algorithm that solves Partially Embedded Planarity for biconnected graphs. This algorithm uses complex subprocedures that handle more restricted subcases. For the connected and disconnected cases, they then show that the additional compatibility constraints can also be tested in linear time. While their work constitutes a significant result, the algorithm described therein is highly complex, its description spans more than 30 pages. Even when the graph H that comes with a fixed drawing is connected; i.e., the embedding is uniquely determined by the rotation system of G, it uses a decomposition of G first into its biconnected, and then into its triconnected components. When H is not connected, these algorithms are applied for each face of H. The resulting algorithm is thus highly technical and relies on a large number of non-trivial subprocedures and data structures. It is therefore not surprising that no implementation is available to date.

In this chapter, we propose an alternative solution for the problem, which is independent of the work of Angelini et al. The algorithm itself straightforwardly extends the well-known vertex-addition planarity test of Booth and Lueker [BL76] (or rather its generalization by Haeupler and Tarjan [HT08] that is presented in Chapter 4). The core of our approach is a modification of the underlying data structure, the PC-tree, that allows it to additionally handle the constraints that stem from the partial drawing. Altogether, this yields a strongly simplified algorithm that relies on depth-first search together with a single non-trivial data structure.

An extensive summary of the underlying vertex-addition planarity test is given in Chapter 4. This chapter is organized as follows. Section 5.1 contains the description of our algorithm for Partially Embedded Planarity as well as its proof of correctness. The following Section 5.2 gives the necessary details for a linear-time implementation of our algorithm. Even including the extensive summary of the basic concepts used, the description of our approach is roughly only half as long as the work by Angelini et al. [Ang+15b] while at the same time being far less technical.

5.1 Partially Embedded Planarity

Recall that in the Partially Embedded Planarity problem we are given an instance $(G, H, \mathcal{H})$ where G is a graph with a subgraph H and $\mathcal{H}$ is an embedding of H. We seek a planar embedding $\mathcal{G}$ of G whose restriction to H coincides with $\mathcal{H}$. In their solution, Angelini et al. [Ang+15b, Lemma 3.9] ensure this condition by enforcing

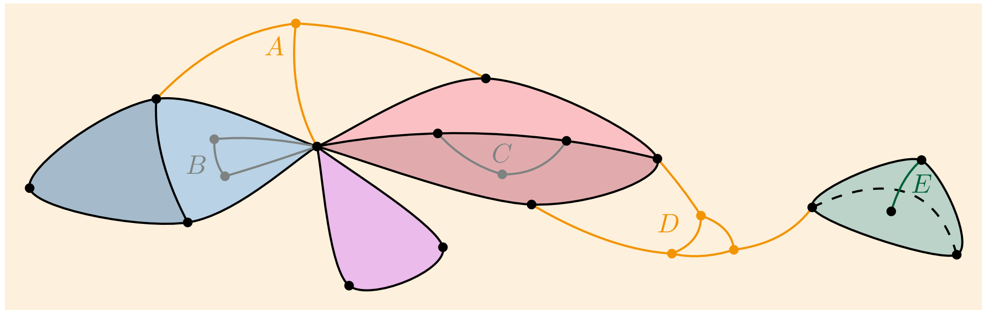


Figure 5.1: A positive instance of Partially Embedded Planarity. The graph H is shown with bold, black edges. The graph $G - H$ is divided into 5 bridges A, B, C, D and E. Bridges B and C have all their attachments in a single block and are thus not restricted to a face. Bridges A, D, and E have attachments in different blocks and are thus restricted to a face. Adding the dashed edge to H would turn the instance negative, as the attachments of bridge E would no longer share a face.

that (i) around each vertex v of H the cyclic order of the edges of H is the same in $\mathcal{G}$ and in $\mathcal{H}$ and (ii) for each (directed) facial cycle of $\mathcal{H}$, the vertices of H that are embedded left and right of it coincide in $\mathcal{G}$ and in $\mathcal{H}$. It is the second condition, also referred to as having correct relative positions, that is relatively complicated to handle efficiently. Note that, if G is connected, the embedding of G and also the induced relative positioning of the connected components of H can be expressed solely in terms of the rotation system of G. Angelini et al. [Ang+15b, Theorem 4.14] give a simple reduction from the case of a non-connected G to solving the different connected components of G independently. We thus limit our attention to the case where G is connected.

Consider two components C_1, C_2 of H connected by a path p in $G - V(H)$. The fact that p does not contain any vertices of H except for its endpoints already ensures that C_1, C_2 are embedded on the same side of any facial cycle of any other component of $\mathcal{H}$. It remains to ensure that C_2 is embedded in the correct face of C_1 and vice versa. Thanks to the connectivity of G, ensuring this for each pair C_1, C_2 of components of H connected by a path p in $G - V(H)$ is sufficient to ensure correct relative positions. Note that not only p has to be in a certain face of $\mathcal{H}$, but this also applies to the whole connected component of $G - V(H)$ that contains p, which we call an H-bridge. Formally, an *H-bridge* B is either a single edge $e \in E(G) \setminus E(H)$ with both end-vertices in H or a connected component B of $G - V(H)$; see Figure 5.1. The *attachments* of a connected component B are the vertices of H whose removal disconnects B from the remaining graph or, in the case of a single edge B, its endpoints. Note that each H-bridge of G has to lie in exactly

one face of $\mathcal{H}$ as it contains no vertices of H. If an H-bridge B has attachments in at least two distinct blocks of H, then that face is uniquely determined; see [Ang+15b, Section 2.3] and Figure 5.1. Thus, we can color each edge $e \in E(G) \setminus E(H)$ with the unique face $f(e)$ of $\mathcal{H}$ in which e must be embedded, or e is uncolored as this face is arbitrary and we set $f(e) = \perp$. Angelini et al. give a simple algorithm that computes this coloring in linear time [Ang+15b, Lemma 2.2].

To ensure correct relative positions, we now need to ensure that at every vertex all edges to incident H-bridges are embedded in the correct face of $\mathcal{H}$. Furthermore, we need to respect the cyclic order of incident edges of H as given in $\mathcal{H}$, where the angles between these edges correspond to faces of $\mathcal{H}$. We will express both requirements as constraints on the rotation of the vertex, interpreting the face assignment as a coloring.

Let v be a vertex of G. A *color constraint* $C_v = (F_v, R_v, U_v, \rho_v, f)$ for v partitions the edges incident to v into a set $F_v = E(v) \cap E(H)$ of *fixed edges*, a set $R_v = \{e \in E(v_1) \setminus E(H) \mid f(e) \neq \perp\}$ of *restricted edges*, and a set $U_v = \{e \in E(v_1) \setminus E(H) \mid f(e) = \perp\}$ of *unrestricted edges*. The fixed edges have a fixed counter-clockwise order $\rho_v = \mathcal{H}(v)$, in which we want to insert the remaining edges to find a rotation for v. For our purposes, we need to additionally constrain where the restricted edges can be inserted. For each fixed edge h, $\mathcal{H}$ defines a face $f_v(h)$ which is incident to h in counter-clockwise direction around v. A *valid cyclic order* σ of the edges incident to v is obtained by arbitrarily inserting the restricted and unrestricted edges of v into ρ_v in such a way that, for each restricted edge e with (in counter-clockwise order) preceding fixed edge h, $f(e) = f_v(h)$ holds. If $F_v = \emptyset$, any order of $R_v \cup U_v$ is valid. Figure 5.2 shows an example of a color-constraint, where for example the restricted edge e with label 5 and the fixed edge h with label 4 have $f(e) = f_v(h)$ represented through the purple color.

In the following Sections 5.1.1 and 5.1.2, we show how such constraints can also be represented using augmented PC-trees and how our operations work on these PC-trees. In Section 5.1.3 following thereafter, we describe how these PC-trees can be used to test Partially Embedded Planarity on biconnected graphs. Section 5.1.4 shows how to lift this requirement and also test non-biconnected instances.

5.1.1 Color-Constrained PC-Trees

To test Partially Embedded Planarity using the PC-tree-based planarity test described in Section 4.2, we introduce a variant of the PC-tree that can also encode these color-constraints from the graph. A *color-constrained PC-tree* has three different types of nodes:

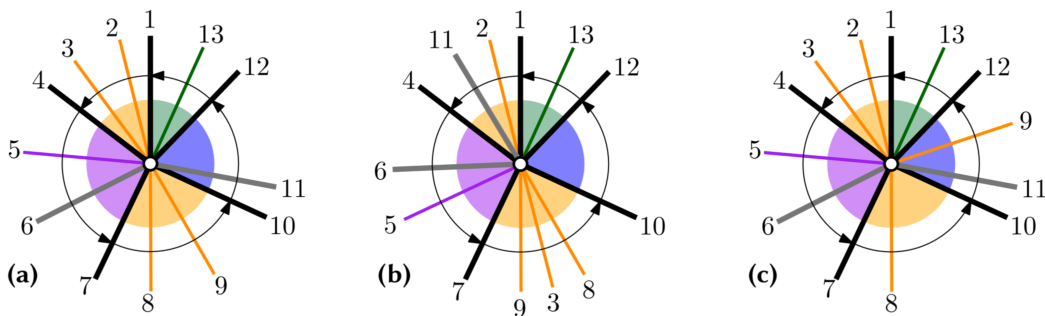


Figure 5.2: Example of vertex with a color-constraint. The fixed edges are fat and black, their fixed counter-clockwise order is also indicated by arrows. The faces following the fixed edges are indicated by colored angles. Each restricted edge is drawn using the respective color of its face, it may be inserted into an arbitrary angle of the same color. **(a)** and **(b)** show two different valid cyclic orders orders around v. The order in **(c)** is not valid, since the orange edge with label 9 is in the blue angle between 10 and 12.

- C-nodes, which behave exactly as in the case of normal PC-tree,
- *fixed C-nodes*, for which the order of their incident edges is completely fixed and may not even be reversed, and finally
- *color-constrained P-nodes*, which are P-nodes with a color-constraint as defined for graph vertices above.

When the context is clear, we refer to the latter simply as P-nodes. Note that an ordinary P-node μ is a special case of a color-constrained P-node with a color-constraint where $F_\mu = R_\mu = \emptyset$. Similarly, a color-constrained P-node with a color-constraint where $R_\mu = U_\mu = \emptyset$ is equivalent to a fixed C-node.

As with usual PC-trees, choosing a valid cyclic order of the edges incident to each inner node of a color-constrained PC-tree T determines a cyclic order of its leaf set $L(T)$. Therefore also a color-constrained PC-tree T represents a set $\omega(T)$ of cyclic order of its leaves. To use such trees in the vertex-addition planarity test, we extend the different operations of PC-trees to also respect the color-constraints and provide updated definitions of the functions we used in our generic planarity testing algorithm. The operations `Merge` and `Split` as described in Section 4.1 can be easily implemented in an analogous fashion to PC-trees. The main operation of interest is the `Update` operation, which given a color-constrained PC-tree T and a subset $A \subseteq L(T)$ of its leaves produces a color-constrained PC-tree $T' = T + A$ with $\omega(T') = \{\sigma \in \omega(T) \mid A \text{ is consecutive in } \sigma\}$. We describe this operation in Section 5.1.2.

The biggest difference between usual and color-constrained PC-trees concerns the operation `Intersect`. In contrast to usual PC-trees, given two color-constrained PC-trees T_1, T_2 on the same set of leaves there generally does not exist a color-constrained PC-tree T with $\omega(T) = \omega(T_1) \cap \omega(T_2)$. For example, if both color-constrained PC-trees consist of a single P-node but with different fixed edges incident, both fixed orders can be interleaved arbitrarily in the intersection, which cannot be represented using a color-constrained PC-tree. This operation is however not strictly needed for the planarity test. Instead, we will only test whether $\omega(T_1) \cap \omega(T_2) \neq \emptyset$ in the context of the planarity test. Conceptually, such a test can be performed using an approach similar to the intersection of ordinary PC-trees and checking for each inner node of the resulting tree whether they allow for at least one common order. Lemma 5.7 shows how to efficiently conduct such a test.

5.1.2 Update Procedure

The `Update` procedure on color-constrained PC-trees is based on the same steps as the update on ordinary PC-trees, although we need to make some modifications to account for the constraints. These modifications can be summarized as follows. The labeling in step (1) works the same as on regular PC-trees. The reordering in step (2) now also needs to respect fixed C-nodes and the order of fixed edges around P-nodes. When splitting nodes in step (3), we need to correctly distribute the constraints across the new nodes and edges resulting from the split. Especially, splitting a node may separate restricted edges from fixed edges and thus also from the angles they want to be embedded in, making the split impossible. Lastly, the contractions in step (4) need to respect fixed C-nodes when merging C-nodes and they need to correctly re-assign the edge constraints when contracting degree-2 nodes. In the following, we will focus only on which changes need to be made to obtain the correct result and defer the details on how these changes can be implemented in amortized linear time to Lemma 5.6.

In step (2) we need to ensure that the full fixed edges (i.e. the incident fixed edges leading to full subtrees) are consecutive in the cyclic order of fixed edges. Furthermore, we need to ensure that both of the at most two terminal fixed edges are directly adjacent to this block of full fixed edges, or adjacent to each other if the block is empty. Finally, we need to check that all P-nodes and all fixed C-nodes have their fixed full edges on the same side of the terminal path. If any of these checks fails, we abort and report an invalid restriction. We will ensure that all restricted full edges are consecutive with the fixed ones in the next step.

When splitting a color-constrained P-node with fixed edges in step (3), we also need to maintain the constraint information and especially ensure that the split parts

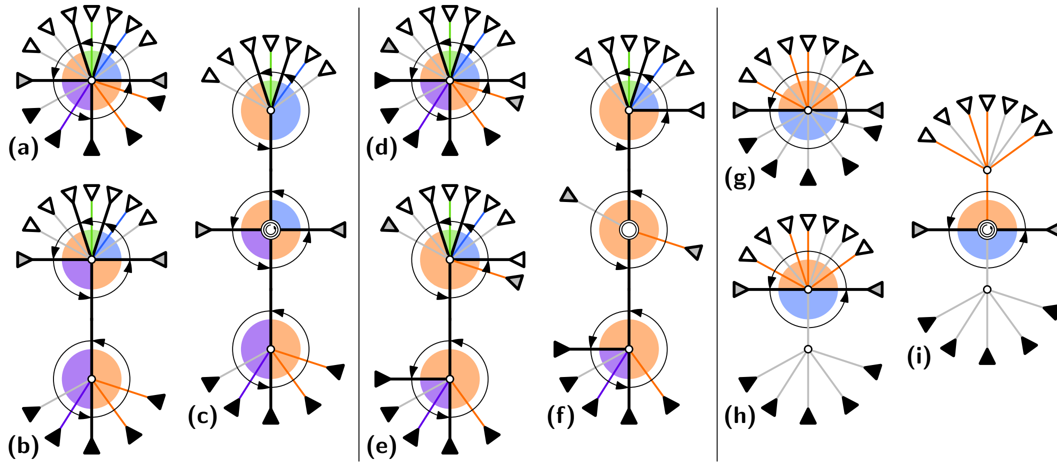


Figure 5.3: Splitting three different color-constrained P-nodes **(a)**, **(d)**, and **(g)**. The first step (**(b)**, **(e)**, and **(h)**) splits off the incident full (black) subtrees, the second step (**(c)**, **(f)**, and **(i)**) splits off the empty (white) subtrees. All splits up to **(f)** have fixed edges on both sides. The split in **(h)** splits off only unrestricted edges, while split **(i)** splits off unrestricted and restricted edges. Only the C-node created in **(f)** is not fixed, as the reversal of its shown rotation is also admissible.

still allow the same relative positions with regard to each other. This especially means that the new edges connecting them need to have the right restrictions assigned. We will describe how to do this when splitting off the full edges F to μ^F, splitting off the empty ones from the resulting node μ' to μ^E works analogously.[1] We make a case distinction based on whether both μ^F and μ' receive at least one fixed edge; see Figure 5.3. If this is the case, edge a between μ^F and μ' is fixed at both ends and the orders of the fixed edges are set to $\rho_{\mu'} = \rho_\mu[(F \cap F_\mu) \rightarrow a]$ and $\rho_{\mu^F} = \rho_\mu[(F^c \cap F_\mu) \rightarrow a]$. All fixed edges retain their color, while a is assigned the color that followed F before the split at μ' and the color that preceded F at μ^F. We need to check that each of the nodes still has at least one appropriate angle for every restricted edge, or abort and report an impossible restriction otherwise. If one of μ^F, μ' received no fixed edges, we assume without loss of generality that μ^F receives all fixed edges as the converse case works analogously. Here, we set $\rho_{\mu^F} = \rho_\mu$ and $\rho_{\mu'} = \emptyset$, leaving the coloring of fixed edges as-is. The restriction of edge a is set according to which edges μ' retained[2]; see Figure 5.3. If there are restricted edges of more than one color at μ', we abort and report an impossible restriction. If there

1 Note that in a linear-time implementation, we cannot process all incident empty edges. In Lemma 5.6, we show that instead splitting off the terminal path edges maintains the time bound.
2 Again, the counters we use to make this distinction in constant time are described in Section 5.2.

are restricted edges of exactly one color c at μ', we set the edge a to be restricted to c at μ^F and to be unrestricted at μ'. If there are no restricted edges at μ', a is unrestricted at both its ends.

If the middle node μ^M resulting from the two splits has degree 4, we need to restrict its order of incident edges such that the terminal path edges are non-adjacent, i.e., the full and empty nodes are on different sides of the terminal path. Note that for a degree-4 node, there are at most two such admissible orders, which are the reverse of each other. If both are allowed by μ^M (which currently is still a color-constrained P-node), we convert μ^M to an ordinary C-node with one of the two orders, otherwise to a fixed C-node with the single possible order; see Figure 5.3.

Finally, we need to ensure that all middle nodes always have the empty and full subtrees, respectively, on the same side of the terminal path. This is done by the contractions of adjacent C-nodes along the terminal path in step (4). The C-node resulting from this is fixed if and only if at least one of its constituent C-nodes was fixed.

▶ **Lemma 5.1.** For a color-constrained PC-tree T and a subset L of its leaves, there exists a color-constrained PC-tree $T' = T + L$ with $\omega(T') = \{\tau \in \omega(T) \mid L \text{ is consecutive in } \tau\}$. ◀

Proof. We will show that the tree T' we obtained by applying our modified `Update` procedure satisfies this condition. Note that T' can be converted into an ordinary PC-tree, which we will refer to as $\texttt{Project}(T')$, by converting all color-restricted P-nodes into ordinary P-nodes (dropping their color constraints) and converting all fixed C-nodes into ordinary C-nodes (possibly now also allowing reversal of their orders). As our modified update make the same changes to the tree structure as in the normal update operation we have $\texttt{Project}(T') = \texttt{Project}(T) + L$ if the restriction is possible. As the projection only allows additional orders, we have $\omega(T') \subseteq \omega(\texttt{Project}(T')) = \omega(\texttt{Project}(T) + L)$, in particular, L is always consecutive in T'.

To show the claimed equivalence of admissible orders, we first show that if $\tau \in \omega(T)$ and L is consecutive in τ then $\tau \in \omega(T')$. Note that the restriction must be possible and T' thus cannot be the null-tree, as an impossible restriction would imply that there is no $\tau \in \omega(T)$ where L is consecutive. As we have $\tau \in \omega(\texttt{Project}(T) + L)$ it is also $\tau \in \omega(\texttt{Project}(T'))$ due to the above equivalence and it remains to show that τ satisfies the color-constraints of T'. To do this, we will apply the changes made by the update procedure to T while maintaining its embedding given by τ to obtain an admissible embedding of T'. First, observe that for all terminal nodes the incident full and empty subtrees with a fixed ordering are respectively consecutive and on different sides of the terminal path, as we would have otherwise returned a

null-tree in step (2). Now consider one of the two splits applied in step (3). Note that the split-off set A needs to be consecutive in the embedding induced on the current P-node μ by order τ as otherwise L would not be consecutive in τ. Thus, the edges in A can be reassigned to a new P-node μ' adjacent to μ in place of A while maintaining the order of A. Note that if A contains no fixed edges, all edges of A were embedded in a single angle. If there are no fixed but restricted edges in A, they all need to have the same color which coincides with the color restricting the edge that replaces A. The conversion of μ^M into a C-node after the two splits can also be done while maintaining the embedding, as the only disallowed rotations of μ^M are those that do not have full and empty subtrees on different sides of the terminal path. Finally, contracting all C-nodes on the terminal path cannot contradict the embedding as we already ensured that all full and empty subtrees are on the correct sides of the terminal path.

Conversely, we need to show that if $\tau \in \omega(T')$ then also $\tau \in \omega(T)$. From our initial considerations it follows that we have $\tau \in \omega(\texttt{Project}(T')) = \omega(\texttt{Project}(T) + L)$ and it again remains to show that τ satisfies the colors constraints of T. To do this, we undo the changes made by the update procedure that turned T into T' while maintaining its embedding given by τ to obtain an admissible embedding of T. As all changes only restrict the number of admissible embeddings, undoing them cannot turn an admissible embedding invalid. ∎

5.1.3 Testing Biconnected Partially Embedded Graphs

We now show how to combine color-constrained PC-trees with the vertex-addition planarity test from Section 4.2 to test biconnected instances of Partially Embedded Planarity. Let $(G, H, \mathcal{H})$ be a partially embedded graph where G is biconnected. Let $v_1, \dots, v_n$ again be an st-ordering of the vertices of G. Recall that for $i = 1, \dots, n$ we have $V_i = \{v_1, \dots, v_i\}$, $G_i = G[V_i]$ and G_i^+ is the graph obtained from $G[V_i]$ by adding all half-embedded edges as half-edges. Also recall that the st-ordering ensures that both $G[V_i]$ and $G[V \setminus V_i]$ are connected. We set $H_i = H[V_i \cap V(H)]$, define H_i^+ to be the graph obtained from H_i by adding all half-embedded fixed edges as half-edges, and define $\mathcal{H}_i^+$ as the restriction of $\mathcal{H}$ to H_i^+. Let $\mathcal{H}_i^\oplus$ be a topological drawing of $\mathcal{H}_i^+$ inside a disk whose boundary visits the non-vertex endpoints of the half-edges according to their order on the outer face.

Now consider an embedding $\mathcal{G}$ that is a solution for $(G, H, \mathcal{H})$ and for a $i \in \{1, \dots, n\}$ a partial solution $\mathcal{G}_i^+$ that is a restriction of $\mathcal{G}$ to G_i^+. Analogously to $\mathcal{H}_i^\oplus$, we define $\mathcal{G}_i^\oplus$ to be a topological drawing of $\mathcal{G}_i^+$ on a disk with all half-edges ending at its boundary. Note that each face of $\mathcal{G}_i^\oplus[H_i^+]$, which is the restriction of $\mathcal{G}_i^\oplus$ to H_i^+, corresponds to a distinct face of $H_i^\oplus$. The embedding $\mathcal{G}_i^+$ that is a partial solution satisfies the following three properties:

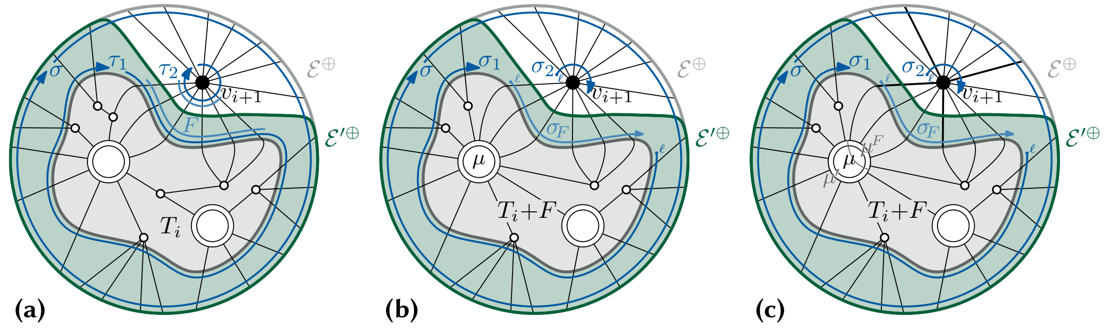


Figure 5.4: (a) Copy of Figure 4.3 (b). The PC-tree T_i representing all planar embeddings of V_i. Relevant orders of half-edges are marked in blue. **(b)** Copy of Figure 4.3 (c). The PC-tree $T_i + F$ with a C-node μ. **(c)** Splitting $T_i + F$ tree also splits μ into μ' in $(T_i + F)[F \to \ell]$ and μ^F in $(T_i + F^c)[F^c \to \ell]$. The edges incident to μ^F are those in $E(T_i, F)$. If the bold edges of v_{i+1} are considered fixed, they fix the flip of μ^F and thus also of μ and μ'.

(E1) All half-edges lie on the outer face.

(E2) Drawing $\mathcal{G}_i^{\oplus}[H_i^+]$ coincides with $\mathcal{H}_i^{\oplus}$.

(E3) Each edge $e \in E(G_i^+) \setminus E(H)$ with $f(e) \neq \bot$ is embedded in a face of $\mathcal{G}_i^{\oplus}[H_i^+]$ that corresponds to $f(e)$ in $\mathcal{H}$.

We are interested in the embeddings of G_i^+ that satisfy these conditions. Let $\Omega_{\mathcal{H}}(G_i^+)$ be the set of all embeddings of G_i^+ that satisfy properties (E1)–(E3), and let $\omega_{\mathcal{H}}(G_i^+) = \{\omega(\mathcal{E}) \mid \mathcal{E} \in \Omega_{\mathcal{H}}(G_i^+)\}$ contain all orders of half-edges on the outer face from these embeddings. Note that any planar embedding $\mathcal{G}_n^+$ (which does not contain any half-edges) that satisfies these properties, or especially property (E2), is a solution for the partially embedded graph. Thus, these properties are necessary for all considered subgraphs, but also sufficient for the full graph. We are now ready to describe our algorithm.

To test an instance $(G, H, \mathcal{H})$ of partially embedded planarity, we compute color-constrained PC-trees $T_1, \ldots, T_n$ satisfying the invariant $\omega_{\mathcal{H}}(G_i^+) = \omega(T_i)$ for all $i \in \{1, \ldots, n\}$. The tree T_1 consist of a single P-node μ with leaves $E(v_1)$ and constraints copied from v_1.

Given a color-constrained PC-tree T_i, the next color-constrained PC-tree T_{i+1} is obtained as follows. We turn v_{i+1} into a color-constrained PC-tree S that consists of a single P-node with leaves $E(v_{i+1})$ and constraints copied from v_{i+1}. Now, we make the edges F between G_i and v_{i+1} consecutive in T_i and S using the `Update` operation. We again split the resulting $T_i + F$ into T^F and T' as in the unconstrained setting, where T^F describes the order of half-edges F leading to v_{i+1} and T' describes the

order of the remaining half-edges. Similarly, we split $S + F$ into S^F and S', where S^F describes the order of half-edges F leading from v_{i+1} to V_i and S' describes the order of the remaining half-edges.

There are two important differences to the ordinary planarity algorithm. First, the orders of half-edges around v_{i+1} are not only constrained by T^F, but also by v_{i+1}, that is by the color-constrained PC-tree S^F. We need to check that T^F and S^F allow for at least one common order of F, that is whether $\omega(T^F) \cap \overline{\omega(S^F)} \neq \emptyset$ as the order of edges in F entering v_{i+1} is the reversal of the order in which they leave G_i^+. Second, if $X = E(T_i + F, F)$ is a set of edges that are consecutive around a C-node μ of $T_i + F$, the constraints of v_{i+1} may fix the order of X around μ and thus the flip of μ; see Figure 5.4 (c). Note that splitting T_i then splits μ into a C-node μ^F in T^F and a C-node μ' in T', both incident to the leaf ℓ introduced by the split. Both μ^F and μ' need to be fixed if the order of X around μ is fixed by v_{i+1}. We detect and handle this case as follows. After finding one order in $\omega(T^F) \cap \overline{\omega(S^F)}$ in the intersection test, we check whether the intersection also contains a second order where μ^F is flipped the other way. If this is not the case, v_{i+1} fixes μ and we accordingly fix the flip of μ' in the copy T'' of T'; otherwise we set T'' to be equal to T'. Finally, we merge the trees S' and T'' at ℓ as before to obtain T_{i+1}.

In our algorithm and especially its following proof of correctness, we use three assumptions regarding the trees generated by the algorithm:

(T1) If the instance is positive, for each step i, the leaves F can be made consecutive in T_i and S.

(T2) The graph G_i^+ that T_i represents is connected.

(T3) The leaves F are all adjacent to the same P-node in S.

Note that these three assumptions are trivially satisfied in the biconnected case we currently investigate. Furthermore note that we have $L(T_n) \setminus F = \emptyset$ in the last step of our algorithm, a situation which will also appear more often throughout the algorithm in the non-biconnected case. This no problem though, as we will never assume $L(T_n) \setminus F \neq \emptyset$ in our proof of correctness.

▶ **Lemma 5.2.** For every step $i \in \{1, \ldots, n\}$ of the algorithm, $\omega_{\mathcal{H}}(G_i^+) = \omega(T_i)$ holds. ◀

Proof. We prove this by induction on the number of steps. For step $i = 1$, observe that T_1 by construction allows the same rotations as v_1. Thus, $\omega_{\mathcal{H}}(G_1^+) = \omega(T_1)$ holds. For the inductive step, assume that $\omega_{\mathcal{H}}(G_i^+) = \omega(T_i)$ holds for step i. We will show the statement for the next step $i + 1$ by arguing both inclusions separately.

Direction $\subseteq$. We first show $\omega_{\mathcal{H}}(G^+_{i+1}) \subseteq \omega(T_{i+1})$. Let $\sigma \in \omega_{\mathcal{H}}(G^+_{i+1})$ and let $\mathcal{E} \in \Omega_{\mathcal{H}}(G^+_{i+1})$ be a corresponding embedding with $\omega(\mathcal{E}) = \sigma$. Let $\mathcal{E}'$ be the embedding of G^+_i obtained by deleting v_{i+1} together with its incident half-edges from $\mathcal{E}$, turning incident ordinary edges to half-edges; see Figure 4.3 (a). As $\mathcal{E} \in \Omega_{\mathcal{H}}(G^+_{i+1})$, it satisfies properties (E1)–(E3). Note that due to property (E1), v_{i+1} must be on the outer face if it has half-edges. If it has none, we have $i + 1 = n$, G^+_{i+1} contains no half-edges, and we can thus choose an arbitrary face incident to v_n to be the outer one. Removing v_{i+1} and turning its incident edges into half-edges thus leaves all half-edges on the same face, that is the outer one, and $\mathcal{E}'$ thus satisfies property (E1). As $\mathcal{E}$ satisfies property (E2), its restriction $\mathcal{E}'$ also does so. Regarding property (E3), we consider three different types of faces of $\mathcal{H}$ that are present in $\mathcal{E}^\oplus$. Faces that are not incident to v_{i+1}, together with all edges of $G - H$ they contain, remain unchanged in $\mathcal{E}'^\oplus$, thus these edges still satisfy property (E3). Faces that are incident to v_{i+1} and no vertex from V_i are incident to the border of $\mathcal{E}^\oplus$ and only contain half-edges with v_{i+1} as endpoint. These faces together with all their contained edges are removed in $\mathcal{E}'^\oplus$ and these edges can thus not violate property (E3). Lastly, consider the set $\mathcal{F}$ of faces that are incident to v_{i+1} as well as a vertex from V_i. At most two of these may also be incident to the border of $\mathcal{E}^\oplus$, while the remaining ones are closed by v_{i+1} in $\mathcal{E}^\oplus$. Note that in case $v_{i+1} \notin V(H)$, we have $|\mathcal{F}| = 1$ as v_{i+1} and its incident edges must lie entirely within one face of $\mathcal{H}$. In either case, all faces of $\mathcal{F}$ are also present in $\mathcal{E}'^\oplus$. The edges between v_{i+1} and V_i turn into half-edges, the half-edges only incident to v_{i+1} are removed, while the half-edges incident to V_i are retained. As the assignment of these (half-)edges to faces remains unchanged, property (E3) is satisfied also in this last case.

As all three properties are satisfied in $\mathcal{E}'$, we thus have $\mathcal{E}' \in \Omega_{\mathcal{H}}(G^+_i)$. As G_i is connected by assumption (T2) (the st-ordering ensures this), we can define $\tau_1 = \omega(\mathcal{E}')$. Similarly, let τ_2 be the order of edges (including half-edges) incident to v_{i+1} in $\mathcal{E}$; see Figure 5.4 (a). As all half-edges are on the outer face of $\mathcal{E}$ and $\mathcal{E}'$, F is consecutive both in τ_2 and in τ_1. Observe that $\tau_1[F] = \overline{\tau_2[F]}$. Since $\mathcal{E}$ can be obtained by combining $\mathcal{E}'$ with v_{i+1} embedded according to τ_2, σ can be obtained by merging τ_1 and τ_2 at F, that is $\sigma = \sigma_1 \otimes_\ell \sigma_2$ for $\sigma_1 = \tau_1[F \to \ell]$ and $\sigma_2 = \tau_2[F \to \ell]$; see Figure 5.4 (b).

As $\mathcal{E}' \in \Omega_{\mathcal{H}}(G^+_i)$ we have $\tau_1 \in \omega_{\mathcal{H}}(G^+_i)$ and, by the inductive hypothesis, $\tau_1 \in \omega(T_i)$. All edges in F must be consecutive in τ_1 and we thus have $\tau_1 \in \omega(T_i + F)$. As $\sigma_1 = \tau_1[F \to \ell]$, it follows that $\sigma_1 \in \omega(T')$ with $T' = (T_i + F)[F \to \ell]$. Note that we have $\tau_2 \in \omega(S)$ by construction of S. As above, all edges in F must be consecutive in τ_2 and thus $\tau_2 \in \omega(S + F)$. As $\sigma_2 = \tau_2[F \to \ell]$, it follows that $\sigma_2 \in \omega(S')$ with $S' = (S + F)[F \to \ell]$. Recall that $T_{i+1} = T'' \otimes_\ell S'$ where either $T'' = T'$ or T'' is

obtained from T' by fixing the flip of the C-node μ' adjacent to ℓ. If $T' = T''$ we directly have $\sigma = \sigma_1 \otimes_\ell \sigma_2 \in \omega(T' \otimes_\ell S') = \omega(T'' \otimes_\ell S') = \omega(T_{i+1})$ as claimed due to $\sigma_1 \in \omega(T')$ and $\sigma_2 \in \omega(S')$. Otherwise, we have $\omega(T' \otimes_\ell S') \supseteq \omega(T'' \otimes_\ell S')$ and to show $\sigma_1 \otimes_\ell \sigma_2 \in \omega(T'' \otimes_\ell S')$ it suffices to show $\sigma_1 \in \omega(T'')$. That is, the flip that σ_1 induces on μ' coincides with the flip of μ dictated by v_{i+1}; see Figure 5.4 (b). Note that the former is the same as the flip of μ induced by τ_1, while the latter is the same as the flip of μ^F induced by $\overline{\tau_2[F^c \to \ell]}$. As $\tau_1[F^c \to \ell] = \overline{\tau_2[F^c \to \ell]}$ and the projection of τ_1 does not change the flip of μ it induces, both flips have to be the same. This concludes the proof for $\omega_{\mathcal{H}}(G^+_{i+1}) \subseteq \omega(T_{i+1})$.

Direction $\supseteq$. Conversely, we show $\omega_{\mathcal{H}}(G^+_{i+1}) \supseteq \omega(T_{i+1})$. Let $\sigma \in \omega(T_{i+1})$ and recall that, with $T' = (T_i + F)[F \to \ell]$ and $S' = (S + F)[F \to \ell]$, we have $\omega(T_{i+1}) \subseteq \omega(T' \otimes_\ell S')$, where equality holds if we did not fix the flip of μ'. Let $\sigma_1 \in T'$ and $\sigma_2 \in S'$ be compatible orders such that $\sigma = \sigma_1 \otimes_\ell \sigma_2$; see Lemma 4.1 and Figure 5.4 (b). With $T^F = (T_i + F^c)[F^c \to \ell]$ and $S^F = (S + F^c)[F^c \to \ell]$, let $\sigma_F \in \omega(T^F) \cap \overline{\omega(S^F)}$ be an order of $F \cup \{\ell\}$ where the induced flip of μ^F in T^F coincides with the flip of μ' induced by σ_1 in T'. Such an order has to exist as the intersection test either found orders for both flips of μ^F or otherwise the flip of μ' was fixed to the one of μ^F. We set $\tau_1 = \sigma_1 \otimes_\ell \sigma_F$ and $\tau_2 = \sigma_2 \otimes_\ell \overline{\sigma_F}$, see Figures 5.4 (a) and 5.4 (b). If $E(T_i, F)$ is a single edge, we directly get $\tau_1 \in \omega(T' \otimes_\ell T^F) = \omega(T_i + F) \subseteq \omega(T_i)$ due to Lemma 4.1 as $\sigma_1 \in \omega(T')$ and $\sigma_F \in \omega(T^F)$. Otherwise, $\tau_1 \in \omega(T_i + F)$ only holds if the induced flips of μ' and μ^F correspond to the same flip of μ. As we chose σ_F to satisfy this, we also get $\tau_1 \in \omega(T_i)$ in this case. Furthermore, we always have $\tau_2 \in \omega(S + F) \subseteq \omega(S)$ as $E(S + F, F)$ is a single edge. This is because it follows from assumption (T3) that the leaves F are all adjacent to the same P-node in S (which is in this case the only inner node of S).

By the inductive hypothesis, we have $\tau_1 \in \omega_{\mathcal{H}}(G^+_i)$ and there exists an embedding $\mathcal{E}' \in \Omega_{\mathcal{H}}(G^+_i)$ with $\omega(\mathcal{E}') = \tau_1$. We choose τ_2 as rotation for v_{i+1} and add it to $\mathcal{E}'$ to obtain an embedding $\mathcal{E}$ of G^+_{i+1}. Doing so, we effectively complete the half-edges of $\mathcal{E}'$ that are in F by connecting them to v_{i+1} and insert the remaining edges of v_{i+1} as new half-edges. Regarding the orders of these edges, recall that F is consecutive but oppositely ordered in τ_1 and τ_2. This ensures that $\mathcal{E}$ is planar and has all half-edges on the outer face, that is property (E1) is satisfied. Since $(\sigma_1 \otimes_\ell \sigma_F)[F \to \ell] = \sigma_1$ and similarly $(\sigma_2 \otimes_\ell \sigma_F)[F \to \ell] = \sigma_2$, we thus get

$$\begin{aligned}\omega(\mathcal{E}) &= \tau_1[F \to \ell] \otimes_\ell \tau_2[F \to \ell] \\ &= ((\sigma_1 \otimes_\ell \sigma_F)[F \to \ell]) \otimes_\ell ((\sigma_2 \otimes_\ell \overline{\sigma_F})[F \to \ell]) \\ &= \sigma_1 \otimes_\ell \sigma_2 = \sigma,\end{aligned}$$

as order of half-edges on the outer face. To show $\sigma \in \omega_{\mathcal{H}}(G^{+}_{i+1})$, it remains to show properties (E2) and (E3). Both are satisfied in $\mathcal{E}'$ by the inductive hypothesis and in the σ_2-induced embedding of v_{i+1} by construction. For their combination, we distinguish two cases depending on whether v_{i+1} is part of H or not. If $v_{i+1} \notin V(H)$, it must lie entirely within one face of $\mathcal{H}$. Note that in this case, also all edges incident to v_{i+1} are not in H and thus must lie within this same face of $\mathcal{H}$. In particular, this holds for the edges in F. If $f(e) = \perp$ for one $e \in F$, this holds for all edges incident to v_{i+1} and property (E3) cannot be violated by any of the added edges. Otherwise, property (E3) holding for $\mathcal{E}'$ already ensures that e is embedded in face $f(e)$. As v_{i+1} lies in the interior of the face $f(e)$, all its remaining edges are thus also embedded in the same, correct face, and property (E3) is satisfied. As $v_{i+1} \notin V(H)$, adding it does not affect the restriction to H considered by property (E2), which is thus also left satisfied. Thus, all three properties are satisfied if $v_{i+1} \notin V(H)$.

Now consider the case $v_{i+1} \in V(H)$. Note that all edges of $G - V(H)$ present in $\mathcal{E}'^{\oplus}$ still lie in the same face, leaving property (E3) unchanged. Consider the newly-inserted half-edges of $G - V(H)$ that lie in a newly-created face incident to v_{i+1} as well as the border of $\mathcal{E}^{\oplus}$, but not to V_i. For these edges, the order chosen by $\tau_2 \in \omega(S)$ ensures that property (E3) is satisfied. The remaining newly-inserted half-edges lie in one of the at most two faces incident to v_{i+1}, V_i and (two distinct segments of) the border of $\mathcal{E}^{\oplus}$, which we call boundary faces. Here, we distinguish whether F contains an edge that is also in H. If this is not the case, all edges of F lie in the same face of $\mathcal{H}$, which is also the single boundary face. Note that for any edge, the faces incident to the left and right of its one end need to be the same than the faces incident to the left and right, respectively, of its other end. This ensures that both V_i and v_{i+1} agree on the face in which F should be embedded and property (E3) is satisfied. If F contains at least one edge that is also part of H, inserting v_{i+1} may close some faces of $\mathcal{H}$. Note that all edges contained in these faces satisfy property (E3) in $\mathcal{E}'^{\oplus}$ and also do so in $\mathcal{E}^{\oplus}$, where their incident segment of the border of $\mathcal{E}'^{\oplus}$ was effectively contracted into a single point. These faces may contain no newly-inserted half-edges, and all old half-edges are completed to v_{i+1}. In contrast to this, the up to two boundary faces may contain half-edges completed by v_{i+1} as well as old half-edges of V_i that were not yet completed and newly-inserted half-edges originating from v_{i+1}. The boundary face is also incident to at least one edge that is both in F and in H, which ensures that v_{i+1} and V_i agree on the face in which to embed all these edges, satisfying property (E3).

Regarding property (E2), note that the construction of the rotation for v_{i+1} ensures that the constraints of $\mathcal{H}$ are respected for this newly-inserted vertex. The relative position of v_{i+1} with regard to V_i is only relevant if both are not connected in $\mathcal{H}_i$, that is if F contains no edges of H. In this case, all edges in F are restricted to

be embedded in the face shared by v_{i+1} and V_i. hence, in this case property (E3) ensures the correct relative positions and thus property (E2). As all three properties are satisfied, we get $\sigma \in \omega_{\mathcal{H}}(G^+_{i+1})$ for both $v_{i+1} \notin V(H)$ and $v_{i+1} \in V(H)$. This concludes the proof of $\omega_{\mathcal{H}}(G^+_{i+1}) \supseteq \omega(T_{i+1})$. ■

Again, we can apply this process until we obtain a color-constrained PC-tree T_{n-1} or conclude that the instance is negative otherwise. Note that we here also need to perform the last step of the algorithm to also check that the constraints of v_n are respected. In case of success, an embedding of the tested graph can be generated using the same approach as for the ordinary planarity test [Chi+85]. Interestingly, the interactions between the two halves of a split C-node, which we need to explicitly handle by fixing the one half if the other one is implicitly fixed, are also one of the main concerns of the embedding algorithm by Chiba et al. [Chi+85]. In the following subsection, we will show how our algorithm can be extended to non-biconnected graphs. We defer the details of a linear-time implementation to Section 5.2.

▶ **Theorem 5.3.** An instance $(G, H, \mathcal{H})$ of Partially Embedded Planarity with a biconnected graph G is positive if and only if our algorithm performs all steps $i = 1, \dots, n$ without an update yielding the null-tree or a failing intersection test. ◀

5.1.4 Non-Biconnected Instances

The ordinary planarity test from Section 4.2 can be applied to non-biconnected graphs by simply processing each biconnected component independently. This approach unfortunately cannot directly be applied for Partially Embedded Planarity, as we also need to account for the constraints of cut-vertices. Instead of relying on an involved preprocessing step, we extend our testing algorithm to directly handle non-biconnected inputs.

When applying the planarity test to a non-biconnected instance, we can no longer assume $v_1, \dots, v_n$ to be an st-ordering that ensures that both $G[V_i]$ and $G[V \setminus V_i]$ are connected for every $i = 1, \dots, n$. We will retain the property that at least $G[V \setminus V_i]$ is connected by using a leaf-to-root ordering of a DFS-tree [HT08]. Thus, we can still assume all half-edges to lie on the outer face. But, at every step of the algorithm, we may now have multiple distinct connected components in G^+_i, each represented by their own PC-tree. When inserting a next vertex v_{i+1}, this may now cause previously distinct connected components to merge. Note that this may happen independently of whether v_{i+1} is a cut-vertex in G when v_{i+1} separates multiple subtrees in the DFS-tree. We handle this case by incrementally merging the components $C_1, \dots, C_k$ of G^+_i that are adjacent to v_{i+1} as follows. We consider v_{i+1} as

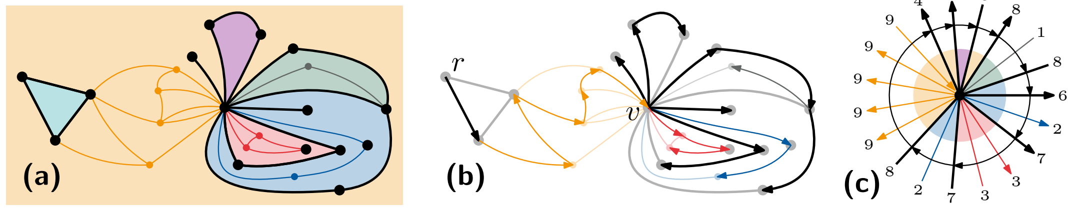


Figure 5.5: (a) An instance of Partially Embedded Planarity with H-bridges colored according to the face they have to be embedded in. **(b)** A DFS tree on the underlying graph G with tree-edges directed away from the root r. **(c)** The color-constrained PC-tree S_0 of vertex v, also indicating the directions of incident edges. The numbers indicate an order in which the incident blocks can be processed. Due to the restricted edges, blocks 3 and 2 need to be removed before blocks 7 and 8, respectively, can be made consecutive. Due to the fixed edges, block 7 needs to be processed before block 8. The given order of blocks with fixed edges is obtain by starting at block 4. Note that the currently shown rotation does not have block 2 consecutive.

initial component $C_0 = \{v_{i+1}\}$ and observe that the union of all components yields the component $C = C_0 \cup C_1 \cup \ldots \cup C_k$ of v_{i+1} in G^+_{i+1}. We compute color-constrained PC-trees $S_1, \ldots, S_k$ satisfying the invariant $\omega(S_j) = \omega_{\mathcal{H}}(G[C_0 \cup \cdots \cup C_j]^+)$ for $j = 1, \ldots, k$. Note that it is $\omega(S_k) = \omega_{\mathcal{H}}(G[C_0 \cup \cdots \cup C_k]^+) = \omega_{\mathcal{H}}(C)$ at the end of the iteration. We obtain tree S_0 with $\omega(S_0) = \omega_{\mathcal{H}}(G[C_0]^+)$ similar to before by converting v_{i+1} into a single P-node and copying its constraints. We use $\mathcal{T}$ to map from components to their already computed color-constrained PC-trees, setting $\mathcal{T}[C] = S_k$ every time we have processed all k components incident to a vertex v_{i+1}. To combine the color-constrained PC-tree $\mathcal{T}[C_{j+1}]$ of the next component C_{j+1} with the current tree S_j into the next tree S_{j+1}, we use the same process as we used for combining T_i with S into T_{i+1} in our test for biconnected instances. Note that while we now run the process on different trees, it can be easily shown that they still satisfy assumptions (T2) and (T3).

It is assumption (T1), that is that the leaves F can always be made consecutive in $\mathcal{T}[C_{j+1}]$ and S_j if the instance is positive, which needs additional consideration. Even if the instance is positive, the update may now fail if the constraints of v_{i+1} require us to nest some incident blocks and we process an outer block before the nested one. Such nesting may be enforced by the fixed order of edges of $\mathcal{H}$ or by the color-constraints around v_{i+1}; see Figure 5.5. Fortunately, we can circumvent this issue as the nested blocks in positive instances need to have a certain structure. Consider a cut-vertex v_{i+1} with an incident block C_{j+1}. When processing C_{j+1}, the component has no further half-edges except those leading to v_{i+1}. We add no new

half edges to S_j and remove all leaves F without replacement after making them consecutive. Thus, no part of $\mathcal{T}[C_{j+1}]$ is present in S_j and we have $L(S_{j+1}) \subsetneq L(S_j)$. This is facilitated by our definition of the merge operation on PC-trees, where merging with a PC-tree that only consists of a single leaf effectively removes the leaf from the other tree. Recall that we never assumed F^c to be non-empty during the proof of Lemma 5.2, thus this does not affect the correctness of our algorithm. If a block now needs to be nested at v_{i+1}, it may not have further half-edges except for those leading to v_{i+1} for the instance to be positive. Thus, processing any nested block before the block is nested within, that is using an inside-out nesting order, ensures any nested blocks are always processed and removed first and the edges of their containing blocks can afterwards be made consecutive. Lemma 5.8 shows how such an order can be found in time linear in the degree of v_{i+1}.

▶ **Lemma 5.4.** Let $C_1, \ldots, C_k$ be the connected components of G_i^+ that are incident to v_{i+1}, ordered according to their inside-out nesting enforced by the constraint of v_{i+1}. For every step $j \in \{0, \ldots, k\}$, $\omega(T_i^j) = \omega_{\mathcal{H}}(G[C_0 \cup \cdots \cup C_j]^+)$ holds. ◀

Proof. The correctness of the statement can be shown analogously to the correctness of Lemma 5.2. To be able to apply this proof, we still need to show that its three underlying assumptions (T1)–(T3) still hold for the new trees we use. Processing the blocks in an inside-out nesting order ensures that, in a yes-instance, blocks that need to be nested are processed before the blocks they are nested within and this process removes all half-edges to the corresponding blocks. This ensures that the half-edges to the outer blocks coming later in the order can also be made consecutive, that is assumption (T1) is fulfilled. As the component C_{j+1} that $\mathcal{T}[C_{j+1}]$ represents is a block incident to v_{i+1}, it is also connected and thus $\mathcal{T}[C_{j+1}]$ satisfies (T2). The fact that the leaves F are all adjacent to the same P-node in S_j, that is that S_j satisfies assumption (T3), can be shown as follows. The assumption holds per construction for S_0. For any later step j with tree S_j, note that the leaves F we make consecutive were already present in S_0, but they where never part of a set we made consecutive in an earlier step. As the PC-tree update only modifies leaves that are made consecutive, the leaves in F are thus all still incident to the same P-node they were incident to in S_0. ■

We can now use these adaptions within the algorithm from the previous section to also test connected but not necessarily biconnected instances. Algorithm 2 illustrates this in pseudocode. There, we use `vertexOrder` to obtain a suitable vertex ordering that leaves $G[V \setminus V_i]$ connected, e.g. by returning the reversed discovery order of a DFS run. Method `vertexToTree` converts a single vertex together with its color-constraints into a color-constrained PC-tree and method `blockOrder`

Algorithm 2: Testing a (not necessarily biconnected) graph G for planarity.

```
1  v_1, ..., v_n ← vertexOrder(G);
2  T ← empty map from connected component to PC-tree;
3  T[{v_1}] ← vertexToTree(v_1);
4  for i in 1, ..., n − 1 do
5      S_0 ← vertexToTree(v_{i+1});
6      C_1, ..., C_k ← blockOrder(v_{i+1});
7      for j in 1, ..., k do
8          F_j ← edges between C_j and v_{i+1};
9          S'_j ← (S_{j−1} + F_j)[F_j → ℓ];   S^F_j ← (S_{j−1} + F_j)[F^c_j → ℓ];
10         if F_j ≠ L(T[C_j]) then
11             T' ← (T[C_j] + F_j)[F_j → ℓ];   T^F ← (T[C_j] + F_j)[F^c_j → ℓ];
12             if ω(T^F) ∩ overline(ω(S^F_j)) = ∅ then
13                 return false
14             if S^F_j fixes C-node μ^F of T^F in the intersection then
15                 T'' ← T' with fixed respective C-node μ';
16             else
17                 T'' ← T';
18             S_j ← S'_j ⊗_ℓ T'';
19         else
20             remove the common leaf e_j from both S^F_j and S'_j;
21             if ω(T[C_j]) ∩ overline(ω(S^F_j)) = ∅ then return false;
22             S_j ← S'_j; // without e_j
23     T[{v_i} ∪ C_1 ∪ ... ∪ C_k] ← S_k;
24 return true if no Update returned the null tree, and false otherwise;
```

yields an inside-out nesting order of the incident blocks. See the following sections for details on how this can be implemented in to run in linear time. To obtain the following result for general instances, we also apply the reduction by Angelini et al. [Ang+15b, Theorem 4.14] that ensures that G is connected.

▶ **Theorem 5.5.** An instance $(G, H, \mathcal{H})$ of Partially Embedded Planarity is positive if and only if our algorithm, for each step $i = 1, \ldots, n$, processes all incident components $C_1, \ldots, C_k$ without an update yielding the null-tree or an intersection test failing. ◀

5.2 Linear-Time Implementation

In this section, we show how the different parts of our algorithm for testing Partially Embedded Planarity can be implemented to run in linear time. Recall that we assume the usual representation of a graph using doubly-linked adjacency lists. We further assume each vertex and edge has a label whether it is contained in H and that the rotation system of $\mathcal{H}$ is given as a separate doubly-linked adjacency list. Additionally, each edge e of H has pointers to objects representing the incident faces at both sides, which we will use as values of $f_v(e)$ and $f_u(e)$ for the endpoints u, v of e. Conversely, the face objects have, for each connected component incident to the face, a pointer to one of their incident edges in the component. Note that we assume that this data structure represents a planar embedding, i.e., we have no cyclically nested faces and components.

For our color-constrained PC-trees we assume a suitable implementation of the underlying PC-tree data structure, which especially allows merging C-nodes in constant time as required for an amortized-linear `Update` [HM03; HM04]; see also Section 8.2. For a linear-time implementation of `Update`, the PC-trees need to be rooted. As a consequence, `Merge` can only be performed in constant time if the leaf ℓ at which we merge is (incident to) the root of at least one of the two trees. Fortunately, the planarity test of Haeupler and Tarjan, on which our algorithm is based, ensures this property [HT08].

Similar to our graph representation, we use a second doubly-linked list to store fixed edges and their cyclic order in the color-constrained PC-trees. To keep track of the colors of the angles following fixed edges and of the restricted edges around a node, we equip each fixed edge (representing its following angle) and restricted edge with a pointer to a shared counter for their node. All objects of the same color at the same node have a pointer to the same counter, which separately counts angles and edges. Note that we do not maintain an index of the colors present at a node, but only an unordered list of all counters present at the node. A counter of a certain

color can thus only be accessed by linearly searching through the counter list, or in constant time via an object of the appropriate color. This is no problem though, as this structure now easily allows decrementing the respective counter when removing an angle or edge from a node. To create new nodes (e.g. when splitting), we keep a single global array with one entry per color that temporary allows looking up counters by color (and subsequently incrementing them appropriately), but only for the currently created node. The counters now allow us to recognize the case when removing the last angle or edge of a color from a node without negatively affecting the asymptotic running time of the `Update` operation.

▶ **Lemma 5.6.** Method `Update` on color-constrained PC-trees can be implemented to run in amortized time linear in $|A|$. ◀

Proof. As the base update procedure for ordinary PC-trees has an amortized running time linear in the number of full leaves [HM03; HM04], we want to show the same also holds for our modified version. Note that to meet the linear time bound, we can only spend a linear amount of time on the full neighbors for each full or partial node, while me may not process all their empty neighbors.

Recall that we leave the labeling in step (1) unchanged. The consecutivity check of step (2) can be implemented by keeping, for each P-node, a linked list of full fixed edges, and checking the predecessor and successor of every such edge after the labeling is complete. For each list, at most one edge may have a non-full predecessor and at most one a non-full successor for the full fixed edges to be consecutive. This also allows us to check that both of the at most two terminal fixed edges are directly adjacent to this block of full fixed edges, or the other partial fixed edge if the block is empty. Similarly, we can check that all P-nodes and fixed C-nodes have their fixed full edges on the same side of the terminal path.

In step (3), we initialize and update the angle and restricted edge color counters appropriately during the splits. These counters then allow us to efficiently detect when a restricted edge got separated from all angles it could be embedded in or whether one of the split halves received no restricted edges. The second split that separates all empty edges is equivalent to splitting off the at most 2 partial neighbors together with the edge leading to the newly-created P-node with all full neighbors, and can thus be implemented in constant time without processing empty edges. After both splits, checking the admissible orders of μ^M and changing its type appropriately can be done in constant time as it has degree at most 4. The contractions in step (4) can then, thanks to the constant-time `Merge` of C-nodes, be done in time linear in the length of the terminal path. ■

In addition to `Update`, we also use a restricted form of the `Intersect` method in our planarity test. Fortunately, this test for a non-empty intersection can easily be implemented in linear time.

▶ **Lemma 5.7.** The test whether $\omega(T^F) \cap \overline{\omega(S^F)} \neq \emptyset$ can be implemented to run in time linear in the number of leaves of S^F and T^F. ◀

Proof. In our implementation, we use that both trees originate from an instance of Partially Embedded Planarity, or, more precisely, that the trees that T^F and S^F stem from satisfy property (E3) and assumption (T3), respectively. On the one hand, this means that T^F already ensures that all restricted edges are embedded in the right angles, i.e., faces. On the other hand, S^F consists of a single P-node. Furthermore, all fixed edges of S^F are also fixed in T^F and have the same incident faces. Similarly, the restricted edges are the same in both trees and they also have the same colors. Thus, the only way to have an empty intersection is if the rotation of the fixed edges of S^F is not admissible by T^F. We can test this by temporarily removing all non-fixed leaves from T^F and checking whether the fixed order of S^F is admissible by the resulting tree. This can easily be checked in linear time, e.g. using an approach similar to the intersection of ordinary PC-trees. ■

Recall that as second modification to the general planarity test, we need to check whether the intersection with the constraints of v_{i+1} fixes the flip of a C-node μ of T_i that is incident to the edges in the set $X = E(T_i + F, F)$. More precisely, we check whether the intersection with S^F fixes the split-off half μ^F of μ in T^F and we therefore also need to fix the other half μ' in T'. This can be checked in linear time by performing the test from Lemma 5.7 twice, once fixing μ^F to its one flip and once to its other flip. We report an empty intersection if both tests fail, fix the flip of μ' accordingly if only one of the two test runs succeeds, and leave μ' unmodified otherwise. Both μ^F and μ' can easily be identified as they are incident to the leaf ℓ introduced by the split.

The last building block we need for our linear-time algorithm is a procedure to find the order we use in Section 5.1.4. Recall that if the added vertex v_{i+1} is a cut-vertex, we need to take special care about its required nesting of incident blocks, as we need to process nested blocks before the blocks they are nested in. Note that v_{i+1} is a cut-vertex if and only if there is at least one component that has no further half-edges except those leading to v_{i+1}, and each such component corresponds to a block around v_{i+1}. In this case, the components with remaining half-edges together with the half-edges of v_{i+1} leading to later vertices form an additional block B_r, as they are all connected via the unembedded, connected graph $G[V \setminus V_i]$. We will ensure that this block comes last in our generated order.

▶ **Lemma 5.8.** In time linear in the degree of vertex v_{i+1} we can find an order $C_1, \ldots, C_k$ of the blocks incident to v_{i+1} such that, whenever the constraints of v_{i+1} require a block C_a to be nested within a block C_b, we have $a < b$. Furthermore, C_k contains all components with remaining half-edges together with the half-edges of v_{i+1} leading to later vertices. ◀

Proof. We will process all blocks without fixed edges first (except for B_r), as these cannot force other blocks to be nested within them. Note that in a yes-instance, a block can only contain restricted edges with different colors if it also contains fixed edges with appropriate incident angles, as the edges could otherwise not be made consecutive. It remains to generate the order of blocks with fixed edges, where the prescribed order of fixed edges $\rho_{v_{i+1}}$ may force blocks to be nested. To do so, we will process the fixed edges in the order of $\rho_{v_{i+1}}$ and keep a stack of blocks for which we have seen some, but not all fixed edges. When encountering the last edge from a block, we remove the block and append it to the processing order of blocks. Note that in a yes-instance, two different blocks may not alternate and we can report a negative instance when we encounter a fixed edge of a block that is within, but not at the top of the stack.

It remains to ensure that the block B_r with the half-edges of v_{i+1} can be put last in the order, which we do by appropriately choosing the edge from which we start the processing of fixed edges in their cyclic order $\rho_{v_{i+1}}$. If B_r contains a fixed edge incident to v_{i+1}, we start processing the cyclic order $\rho_{v_{i+1}}$ with the fixed edge following thereafter. This already ensures that B_r is the last block returned from the stack-based algorithm. If B_r contains no fixed but a restricted edge e incident to v_{i+1}, we start processing from an arbitrary fixed edge following an angle with the same color as e. We then append B_r and note that the choice of first fixed edge ensures that the we still have an appropriately-colored angle available when processing B_r as last block. Lastly, if B_r has only unrestricted edges incident to v_{i+1}, we can start the processing of fixed edges at any point. ■

Altogether, this now allows us to show implement our full algorithm in linear time.

▶ **Theorem 5.9.** An instance $(G, H, \mathcal{H})$ of Partially Embedded Planarity can be tested in time linear in the size of G. ◀

Proof. Our linear-time algorithm for testing Partially Embedded Planarity on general instances works in the following steps. As preprocessing, we first apply two simple linear-time algorithms of Angelini et al. that allow us to process each connected component of G separately and that compute the color constraints $f(e)$

for each edge of an H-bridge [Ang+15b, Theorem 4.14 and Lemma 2.2]. We then run the algorithm described in Section 5.1.4 on each connected component of G. Haeupler and Tarjan [HT08, Section 4] describe how the DFS can be implemented to at the same time yield the separate blocks incident to a vertex v_{i+1} together with the edges connecting them to v_{i+1}. We sort and process the incident blocks using the approach from Lemma 5.8. In the resulting order, we apply the modified algorithm from Section 5.1.3, using the amortized linear-time `Update` from Lemma 5.6. There, we use the intersection check from Lemma 5.7 and, if necessary, fix the respective C-node of the tree T' representing the added block. Note that this is not needed if the current block has no further half-edges, as we can simply remove the respective subtree in this case. Further note that the DFS also ensures that the leaf ℓ at which we merge is the root of T', which allows it to be finally attached to S' in constant time, see [HT08]. This concludes the linear-time implementation of our algorithm. ■

Similar to the linear-time Partially Embedded Planarity solution by Angelini et al. [Ang+15b], the linear-time EC-Planarity solution of Gutwenger et al. [GKM08] also relies on decompositions that allow them to verify constraints in each individual skeleton of the SPQR-tree. Alternatively, both EC-Planarity and the equivalent (non-partial) (F)PQ-constrained Planarity can be solved in linear time via a reduction to Partially Embedded Planarity using the so-called ec-expansion [GKM08; Sch13]. Our simplified approach can also be used to directly solve both problems in linear time without the need for a decomposition or reduction. In both problems, the rotation constraints for each vertex are directly given in the input and no longer need to be computed from a prescribed embedding first. In this case, we only have *FPC-trees* as constraints, that is color-constrained PC-trees that only consist of C-nodes and P-nodes with either only unrestricted or only fixed edges (where the latter are called *F-nodes*), but never restricted edges. Note that we need to handle the intersection test slightly differently, as the fixed edges no longer need to be the same in both trees as edges can now have different restrictions at their endpoints. Fortunately, the intersection of two FPC-trees is still an FPC-tree. It can be found by using the intersection procedure for regular PC-trees while temporarily interpreting F-nodes as C-nodes. Subsequently, any C-node that stems from one or more F-nodes is converted back to an F-node that respects the flip of all its initial F-nodes (or we report an empty intersection if the initial F-nodes do not agree on this flip). We can thus perform the intersection test in linear time and obtain the following corollary.

▶ **Corollary 5.10.** EC-Planarity and (F)PQ-constrained Planarity can be solved in linear time. ◀

5.3 Conclusion

In this chapter, we have given a linear-time solution for the problem Partially Embedded Planarity. Our algorithm straightforwardly extends the well-known vertex-addition planarity test of Haeupler and Tarjan [HT08]. The core of our approach is the modification of its underlying data structure, the PC-tree, to also respect the constraints that stem from the partial drawing. These constraints are derived from the insight that for an extension of the fixed partial drawing, it is necessary and sufficient to respect the fixed vertex rotations as well as embed certain edges in predefined faces between the fixed edges. These constraints can then be naturally translated from the rotations of individual vertices to the PC-trees that represent all planar embeddings of connected components. Switching in these color-constrained PC-trees in the planarity test then requires us to handle two very specific situations more carefully than in the unconstrained setting. We uncovered the assumptions that the unconstrained planarity test makes in these cases and showed how they can be fixed in the constrained setting, which also provides valuable insights into the workings of the generic test.

In comparison to the previous decomposition-based approach by Angelini et al. [Ang+15b], this yields a strongly simplified algorithm that relies on depth-first search together with a single non-trivial data structure. This also allows us to directly model other planarity variants constraining vertex rotations without the need for a prior reduction or decomposition. The description of our algorithm, even when also including the extensive description of its well-known underlying concepts, is roughly half as long as the description of the algorithm by Angelini et al. [Ang+15b], while at the same time being far less technical.

An interesting question for future work is whether our algorithm can be extended to handle constraints to vertex rotations in the form of arbitrary color-constrained PC-trees that in particular were not derived from an instance of Partially Embedded Planarity. The main complication here is that in this case, similar to using arbitrary FPC-trees as constraints in Corollary 5.10, the constraints on an edge no longer need to be the same at both its endpoints.

6 Synchronized Planarity

This chapter is based on joint work with Thomas Bläsius and Ignaz Rutter, which also appeared at ESA 2021 [3] and in the ACM Transactions on Algorithms [1]. In comparison to these versions, Sections 6.4 and 6.5 were extended to show various further applications and related NP-hard problems.

Many variants of constrained planarity come down to the question whether there are embeddings of one or multiple graphs such that the rotations of certain vertices are in sync in a certain way. Consider for example the problem Clustered Planarity (see Section 3.1.4), where the order in which edges "leave" a cluster (via its boundary curve) needs to line up with the order in which they "enter" the cluster. To formulate this in terms of synchronized vertex rotations, we can split the graph into two halves at each cluster boundary: one where we contract the inside of the cluster into a single vertex and one where we contract the outside into a single vertex. Now, the rotations of the two vertices that where obtained from the same cluster boundary need to be in sync. The graph resulting from these separate contractions of each side of a cluster boundary is called CD-tree [BR16a]; see Figure 6.6. A similar construction can also be applied to Level- and Strip Planarity, where the order of edges leaving a horizontal level or strip needs to be the same as the order in which they enter the next one. For the SEFE-2 problem, it directly suffices to find embeddings of both exclusive graphs where the rotations of shared edges are the same as long as the shared graph is connected [BKR17; JS09]; see Figure 6.10. In the Atomic Embeddability problem [FT22], we want to embed a graph without crossings onto a 3-dimensional molecule structure, given a prescribed mapping of vertices to atoms of the molecules. Here, the order in which edges enter a "pipe" connecting two atoms also needs to line up with the order in which they leave it at the other end.

Inspired by these observations, we introduce a new planarity variant. Synchronized Planarity has a loop-free multi-graph together with two types of synchronization constraints as input. Each *Q-constraint* consists of a subset of vertices together with a fixed reference rotation for each of these vertices. The Q-constraint is satisfied if and only if either all these vertices have their reference rotation or all these vertices have the reversed reference rotation. Vertices appearing

in Q-constraints are called *Q-vertices* and all remaining vertices are *P-vertices*.[3] A *P-constraint* between two P-vertices u and v defines a bijection between the edges incident to u and v. It is satisfied if and only if u and v have the opposite rotation under this bijection. We require that the P-constraints form a matching, that is, no vertex appears in more than one P-constraint. The decision problem Synchronized Planarity now asks whether the given graph can be embedded such that all Q- and all P-constraints are satisfied.

Synchronized Planarity serves as a powerful modeling language that lets us express various other planarity variants using simple linear-time reductions. We provide such reductions for Clustered Planarity, Atomic Embeddability, Partially PQ-constrained Planarity, and Simultaneous Embedding with Fixed Edges with a connected shared graph (Connected SEFE-2) and many other constrained planarity variants. Our main contribution is an algorithm that solves Synchronized Planarity, and thereby all the above problems, in quadratic time.

This chapter is organized as follows. In Section 6.1 we give an intuition of the vertex rotation constraints that formed the main obstacle of Clustered Planarity in the past, while discussing previous work on solving this problem in Section 6.2. The main Section 6.3 introduces the Synchronized Planarity problem together with our quadratic solution to it. In Section 6.4 we show how Synchronized Planarity can be used to solve various other problems. We discuss related NP-hard problems in Section 6.5 and give a detailed comparison of our solution to the algorithm by Fulek and Tóth [FT22] in Section 6.6.

6.1 Technical Contribution

Our result has an impact on several planarity variants that have been studied previously. Before discussing this individually in the context of previous publications, we point out a common difficulty that has been a major barrier for all of them and briefly sketch how we resolve it.

Consider the following constraint on the rotation of a single vertex. Assume its incident edges are grouped and we only allow orders where no two groups alternate, that is, if e_1, e_2 are in one group and e_3, e_4 are in a different group, then the cyclic subsequence e_1, e_3, e_2, e_4 and its inverse are forbidden. Such restrictions have been called *partition constraints* before [BR16a], and they naturally emerge at cut-vertices where each incident 2-connected component forms a group. A single

3 The names are based on PQ-trees, where Q- and P-nodes have fixed and arbitrary rotation, respectively. To be consistent with previous publications, we continue to use the naming based on PQ-trees instead of the (equivalent) PC-trees.

partition constraint is not an issue by itself, but it becomes difficult to deal with in combination with further restrictions. In the context of the above example this would, e.g., be a third group e_5, e_6 that must also be non-alternating with the former two. This is why cut-vertices and disconnected clusters are a major obstacle for SEFE-2 [BKR17] and Clustered Planarity [BR16a], respectively.

The same issues appear for Synchronized Planarity, when we have a cut-vertex v that is involved in P-constraints, that is, its rotation has to be synchronized with the rotation of a different vertex u. We deal with these situations as follows, depending on whether u is also a cut-vertex or not. If not, it is rather well understood which embedding choices impact the rotation of u and we can propagate this from u to v.[4] This breaks the synchronization of u and v down into the synchronization of smaller embedding choices, a well-known technique that has been used before [BR16b; GKM08]. If u is also a cut-vertex, we are forced to actually deal with the embedding choices emerging at cut-vertices. This is done by "encapsulating" the restrictions on the rotations of u and v that are caused by the fact that they are cut-vertices. All additional restrictions coming from embedding choices in the 2-connected components are pushed away by introducing additional P-constraints. After this, the cut-vertices u and v have very simple structure, which can be resolved by essentially joining them together. This procedure is formally described in Section 6.3.3 and illustrated in Figures 6.2 and 6.3.

This solution can be seen as a combinatorial perspective on the recent breakthrough result by Fulek and Tóth [FT22], who resolved the cut-vertex issue by applying an idea coming from Carmesin's work [Car17c]. While Carmesin works with 2-dimensional simplicial complexes, Fulek and Tóth achieve their result by transferring Carmesin's idea to the setting of topological graphs on surfaces and combining it with tools from their work on thickenability [AFT19; FT22]. Our work transfers the problem back to an entirely combinatorial treatment of topological graphs in the plane. This further simplification allows us to clearly highlight the key insights that make the algorithm tick and at the same time provides access to a wide range of algorithmic tools for speeding up the computations. The key insights are that the "encapsulation" applies cuts that are respected by all embeddings, that is the constraints we drop are enforced by the resulting graph, see Section 6.3.1, and that each operation not only works towards decreasing the global maximum degree, but also makes clearly-visible progress local to the affected area, see Section 6.3.7. Two main tools which we employ to achieve the running time of our algorithm are the PQ-tree in the form of embedding trees and the SPQR-tree. Using further tools from the planar graphs toolbox, a version of the SPQR-tree that can be maintained under dynamic graph updates [Epp+98] can be used to speed up our algorithm even further, see Section 6.3.8 and Chapter 7.

4 We can also do this if v is not a cut-vertex.

6.2 Related Work

Clustered Planarity was first considered by Lengauer [Len89] and later rediscovered by Feng et al. [FCE95]. In both cases, the authors give polynomial-time algorithms for the case that each cluster induces a connected graph. The complexity of the general problem that allows disconnected clusters has been open for 30 years. In that time, many special cases have been shown to be polynomial-time solvable [AD19; Cor+08; Ful+15; Gut+02] before Fulek and Tóth [FT22] recently settled Clustered Planarity in P. The core ingredient for this is their $O(m^8)$-time algorithm for the Atomic Embeddability problem. It has two graphs G and H as input. Roughly speaking, H describes a 3-dimensional molecule structure with atoms represented by spheres and connections (a.k.a. pipes) represented by cylinders. The other graph G comes with a map to the molecule structure that maps each vertex to the surface of an atom such that two adjacent vertices lie on the same atom or on two atoms connected by a pipe. Atomic Embeddability then asks whether G can be embedded onto the molecule structure according to the given mapping such that no edges cross.

Atomic Embeddability has been introduced as a generalization of the Thickenability problem that appears in computational topology [AFT19]. It can be shown that Atomic Embeddability and Thickenability (and actually also Synchronized Planarity, as discussed in Section 6.6) are linear-time equivalent [FT22]. Thus, the above $O(m^8)$-time algorithm for Atomic Embeddability also solves Thickenability and Synchronized Planarity. In a series of preprints from 2017, Carmesin [Car17a; Car17b; Car17c] gives a Kuratowski-style characterization of Thickenability [Car17a], which he claims yields a quadratic-time algorithm as a byproduct [Car17c]. While it is believable that the running time of his algorithm is polynomial, the correctness of the claimed running time has not been confirmed since. We thus consider the self-contained $O(m^8)$-algorithm by Fulek and Tóth published in 2019, which is seen as an improvement over the algorithm by Carmesin, as state of the art. A detailed comparison of their solution to Atomic Embeddability and our solution to Synchronized Planarity is given in Section 6.6.

To finally solve Clustered Planarity, Fulek and Tóth [FT22] use the reduction of Cortese and Patrignani [CP18] to Independent Flat Clustered Planarity, which they then reduce further to Thickenability. The last reduction to Thickenability is based on a combinatorial characterization of Thickenability by Neuwirth [Neu68], which basically states that multiple graphs have to be embedded consistently, that is, such that the rotation is synchronized between certain vertex pairs of different graphs. Via the reduction from Connected SEFE-2 to Clustered Planarity given by Angelini and Da Lozzo [AD16], the above result extends to Con-

nected SEFE-2, which was a major open problem in the context of simultaneous graph representations [BKR13]. We flatten this chain of reductions by giving a simple linear reduction from each of the problems Connected SEFE-2, Clustered Planarity, and Atomic Embeddability to Synchronized Planarity in Section 6.4, yielding quadratic-time algorithms for all of them. Moreover, the problem Partially PQ-constrained Planarity, for which we also give a linear reduction to Synchronized Planarity, has been solved in polynomial time before, but only for biconnected graphs [BR16b] and in the non-partial setting where either all or none of the edges of each vertex are constrained [GKM08]. Similarly, Row-Column Independent NodeTrix Planarity has been solved in polynomial time for biconnected graphs [LRT21], while our solution works on general graphs.

6.3 The Synchronized Planarity Problem

An instance of the problem Synchronized Planarity is a tuple $I = (G, \mathcal{P}, \mathcal{Q}, \psi)$, where

1. $G = (P \cup Q, E)$ is a (loop-free) multi-graph with P-vertices P and Q-vertices Q,
2. $\mathcal{Q}$ is a partition of Q,
3. ψ is a mapping that assigns a rotation to each Q-vertex, and
4. $\mathcal{P}$ is a set of triples (u, v, φ_{uv}), where u and v are P-vertices of the same degree, φ_{uv} is a bijection between their incident edges, and each P-vertex occurs at most once in $\mathcal{P}$.

We call the triples $\rho = (u, v, \varphi_{uv})$ in $\mathcal{P}$ *pipes*. Pipes are not directed and we identify (u, v, φ_{uv}) and (v, u, φ_{vu}) with $\varphi_{vu} = \varphi_{uv}^{-1}$. We also define $\deg(\rho) = \deg(u) = \deg(v)$. If two P-vertices are connected by a pipe, we call them *matched*; all other P- and Q-vertices are *unmatched*.

The planar embedding $\mathcal{E}$ of G *satisfies the cell* $X \in \mathcal{Q}$ if it is either $\mathcal{E}(v) = \psi(v)$ for all $v \in X$ or $\mathcal{E}(v) = \overline{\psi(v)}$ for all $v \in X$. We say that the embedding satisfies the *Q-constraints* if it satisfies all cells, that is, vertices in the same cell of the partition $\mathcal{Q}$ are consistently oriented. The embedding $\mathcal{E}$ *satisfies the pipe* $\rho = (u, v, \varphi_{uv})$ if $\varphi_{uv}(\mathcal{E}(u)) = \overline{\mathcal{E}(v)}$, that is, they have opposite rotations under the bijection φ_{uv}. We say that the embedding satisfies the *P-constraints* if it satisfies all pipes. The embedding $\mathcal{E}$ is called *valid* if it satisfies the P-constraints and the Q-constraints. The problem Synchronized Planarity asks whether a given instance $I = (G, \mathcal{P}, \mathcal{Q}, \psi)$ admits a valid embedding $\mathcal{E}$ of G.

In the context of Synchronized Planarity, we assume that the embedding tree of a vertex does not allow rotations that would result in a Q-vertex v having any other rotation than its default ordering $\psi(v)$ or its reverse $\overline{\psi(v)}$. To ensure this, we can subdivide each edge incident to v and connect each pair of two of the new nodes if the edges they subdivide are consecutive in the cyclic order $\psi(v)$ [GKM08]. Note that this generates a k-wheel with center v and that there are exactly two planar rotations of the center of a wheel, which are the reverse of each other. Further note that the resulting graph still has $O(m)$ edges. We always generate the embedding trees based on the graph where each Q-vertex in G is temporarily replaced with its respective wheel.

In the following Section 6.3.1 we will first discuss the high-level insight that allows us to solve this problem efficiently. In Sections 6.3.2 to 6.3.5, we will subsequently describe the concrete operations that use this insight to stepwise reduce the degree of individual pipes until they become trivial. Sections 6.3.6 and following consider how long this process takes until all pipes are removed and how and in what time the resulting pipe-free instance can be solved.

6.3.1 Splits and Joins of Graphs and Embeddings

Let $G = (V, E)$ be a graph. We call a partition $C = (X, Y)$ of V into two disjoint cells a *cut* of G. The edges $E(C)$ that have their endpoints in different cells are called *cut edges*. The *split* of G at $C = (X, Y)$ is the disjoint union of the two graphs obtained from two copies of G by contracting X and Y to a single vertex x and y, respectively (keeping possible multi-edges); see Figure 6.1. Note that the edges incident to x and y are exactly the cut edges, yielding a natural bijection φ_{xy} between them. Conversely, given two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ and vertices $x \in V_1, y \in V_2$ together with a bijection φ_{xy} between their incident edges, their *join* along φ_{xy} is the graph $G = (V, E)$, where $V = V_1 \cup V_2 \setminus \{x, y\}$ and E contains all edges of $E_1 \cup E_2$ that are not incident to x or y. Furthermore, for each edge $e = ux \in E_1$ incident to x, the set E contains an edge uv, where v is the endpoint of $\varphi_{xy}(e)$ distinct from y; see Figure 6.1. Observe that split and join are inverse operations.

We say that a planar embedding $\mathcal{E}$ of a graph G *respects* a cut $C = (X, Y)$ if and only if for a topological planar drawing Γ of G with embedding $\mathcal{E}$ there exists a closed curve γ such that (i) γ separates X from Y, (ii) γ crosses each edge in $E(C)$ in exactly one point, and (iii) γ does not cross any edge in $E \setminus E(C)$; see Figure 6.1. We say that γ *represents C in Γ*.

If $\mathcal{E}$ respects C, a split of G at C preserves $\mathcal{E}$ as follows. Let G_1 and G_2 be the graphs resulting from splitting G at C and let $x \in V_1$ and $y \in V_2$ be such that φ_{xy} identifies their incident edges. Let Γ be a topological planar drawing with embedding $\mathcal{E}$ and let γ be a curve in Γ that represents C in Γ. We obtain a planar drawing Γ_1 of G_1 by

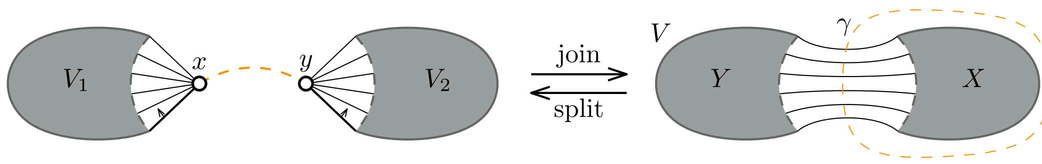


Figure 6.1: Joining and splitting two graphs at $x \in V_1$ and $y \in V_2$. The bijection φ_{xy} between their incident edges is shown as follows: the two bold edges at the bottom are mapped to each other. The other edges are mapped according to their order following the arrow upwards (i.e. clockwise for x and counter-clockwise for y).

contracting to a single point the side of γ that contains V_2, that is, by removing V_2 and connecting the resulting half-edges to a new vertex in place of V_2. Similarly, a planar drawing Γ_2 of G_2 is obtained by contracting the V_1-side of γ to a single point. We denote by $\mathcal{E}_1$ and $\mathcal{E}_2$ the corresponding combinatorial embeddings of G_1 and G_2. Note that by construction for each vertex of $V_1 \setminus \{x\}$ the rotations in $\mathcal{E}$ and $\mathcal{E}_1$ coincide, and the same holds for vertices of $V_2 \setminus \{y\}$ in $\mathcal{E}$ and $\mathcal{E}_2$. Moreover, the rotations $\mathcal{E}_1(x)$ and $\mathcal{E}_2(y)$ are determined by the order in which the edges of $E(C)$ cross γ, and therefore they are oppositely oriented, that is, $\varphi_{xy}(\mathcal{E}_1(x)) = \overline{\mathcal{E}_2(y)}$. We call embeddings $\mathcal{E}_1$ and $\mathcal{E}_2$ with this property *compatible with* φ_{xy}.

Conversely, we can join arbitrary embeddings $\mathcal{E}_1$ of G_1 and $\mathcal{E}_2$ of G_2 that are compatible with φ_{xy} by assuming that x and y lie on the outer face, removing x and y from the embeddings, and connecting the resulting half-edges according to φ_{xy}. The result is a planar embedding $\mathcal{E}$ where for each vertex $v \in V_i \setminus \{x, y\}$ we have $\mathcal{E}(v) = \mathcal{E}_i(v)$ for $i = 1, 2$.

▶ **Lemma 6.1.** Let $G = (V, E)$ be a planar graph and let (X, Y) be a cut of G such that X and Y induce connected subgraphs of G. Then every planar embedding of G respects (X, Y). ◀

Proof. Let $\mathcal{E}$ be a planar embedding of G. Since X and Y induce connected subgraphs, it follows that no proper subset of $E(X, Y)$ is a cut. Therefore, (X, Y) corresponds to a simple cycle $C^\star$ in the dual graph $G^\star$ with respect to $\mathcal{E}$ [Die17, Proposition 4.6.1], which in turn implies that $\mathcal{E}$ respects (X, Y). ■

▶ **Lemma 6.2.** Every planar embedding of a bipartite graph $G = (A \cup B, E)$ respects (A, B). ◀

Proof. Let $\mathcal{E}$ be a planar embedding of G. By [DLR90, Lemmas 3–5] we can augment G and its embedding $\mathcal{E}$ by additional edges in $\binom{A}{2} \cup \binom{B}{2}$ to a graph G' with planar embedding $\mathcal{E}'$ such that A and B induce connected subgraphs. The fact that $\mathcal{E}$ respects the cut (A, B) then follows from Lemma 6.1. ■

6.3.2 High-Level Algorithm

We give an algorithm for solving Synchronized Planarity for graphs with n vertices and m edges in $O(m^2)$ time. Without loss of generality, we assume that G has no isolated vertices and thus $m \in \Omega(n)$. Furthermore, we assume the input graph G to be planar. Our approach hinges on three main ingredients. The first are the three operations EncapsulateAndJoin, PropagatePQ, and SimplifyMatching, each of which can be applied to pipes that satisfy certain conditions. If an operation is applicable, it produces an equivalent instance I' of Synchronized Planarity in linear time. Secondly we show that if none of the operations is applicable, then I has no pipes, and we give a simple linear-time algorithm for computing a valid embedding in this case. The third ingredient is a non-negative potential function ϕ for instances of Synchronized Planarity. We show that it is upper-bounded by $2m$, and that each of the three operations decreases it by at least 1.

Our algorithm is therefore extremely simple; namely, while the instance still has a pipe, apply one of the operations to decrease the potential. Since the potential function is initially bounded by $2m$, at most $2m$ operations are applied, each taking $O(m)$ time. We will show that the resulting instance without pipes has size $O(m^2)$ and can be solved in linear time, thus the total running time is $O(m^2)$.

Conversion of small-degree P-vertices

The main difficulty in Synchronized Planarity stems from matched P-vertices. However, P-vertices of degree up to 3 behave like Q-vertices in the sense that their rotations are unique up to reversal. Throughout this chapter, we implicitly assume that P-vertices of degree less than 4 are converted into Q-vertices, also converting a pipe of degree less than 4 into a Q-constraint with the auxiliary operation ConvertSmall described in the following. We therefore assume without loss of generality that P-vertices, and in particular pipes, have degree at least 4.

Vertices of degree 3 have only two distinct rotations, which are the reverse of each other. Vertices of degree less than 3 have a unique rotation, which coincides with its reverse. We thus define operation ConvertSmall(u, I) to convert a P-vertex u with $\deg(u) < 4$ into a Q-vertex, resulting in an instance $I' = (G', \mathcal{P}', \mathcal{Q}', \psi')$ where $G' = (P' \cup Q', E)$. If u is unmatched, we set $P' = P \setminus \{u\}$, $\mathcal{P}' = \mathcal{P}$, and $Q' = Q \cup \{u\}$ and give u it its own cell in $\mathcal{Q}' = \mathcal{Q} \cup \{\{u\}\}$. We fix an arbitrary order $\psi'(u)$ and let ψ' coincide with ψ for all other vertices. If the P-vertex u is matched with another P-vertex v, we can convert both of them to Q-vertices setting $P' = P \setminus \{u, v\}$, $Q' = Q \cup \{u, v\}$, and $\mathcal{P}' = \mathcal{P} \setminus \{(u, v, \varphi_{uv})\}$. We also put both together

in a cell in $Q' = Q \cup \{\{u,v\}\}$, again setting $\psi'(u)$ as before, but now also defining $\psi'(v) = \overline{\varphi_{uv}(\psi'(u))}$. Note that this enforces that matched vertices u and v have opposite rotations under the bijection φ_{uv}. All other P-vertices and their pipes remain unaffected. Previous Q-vertices already in Q are also unaffected.

▶ **Lemma 6.3.** Applying ConvertSmall to a P-vertex u with $\deg(u) < 4$ yields an equivalent instance in constant time. ◀

Proof. We first show that the conversion preserves a valid embedding $\mathcal{E}$ of I. It is $\mathcal{E}(u) = \psi'(u)$ or $\mathcal{E}(u) = \overline{\psi'(u)}$. If u is unmatched, it is the only Q-vertex in its cell and thus $\mathcal{E}$ also satisfies the Q-constraints of I'. Otherwise, u is matched with P-vertex v and as $\mathcal{E}$ satisfies the pipe uv it is $\varphi_{uv}(\mathcal{E}(u)) = \overline{\mathcal{E}(v)}$. If $\mathcal{E}(u) = \psi'(u)$, we get $\psi'(v) = \overline{\varphi_{uv}(\psi'(u))} = \overline{\varphi_{uv}(\mathcal{E}(u))} = \mathcal{E}(v)$, satisfying the new Q-constraint. The case of $\mathcal{E}(u) = \overline{\psi'(u)}$ follows analogously. As the underlying graph and all other pipes remain unchanged, $\mathcal{E}$ is valid embedding of I'.

Conversely, assume that $\mathcal{E}'$ is a valid embedding for I'. If u is the sole vertex in its cell, converting u to a P-vertex will not affect the validity of the embedding (and also not allow new embeddings as $\deg(u) < 4$). If u shares its cell with vertex v, it is either $\mathcal{E}'(u) = \psi'(u)$ and $\mathcal{E}'(v) = \psi'(v)$ or it is $\mathcal{E}'(u) = \overline{\psi'(u)}$ and $\mathcal{E}'(v) = \overline{\psi'(v)}$. Inserting $\psi'(v) = \overline{\varphi_{uv}(\psi'(u))}$ as chosen shows the constraint $\varphi_{uv}(\mathcal{E}(u)) = \overline{\mathcal{E}(v)}$ of pipe uv is satisfied. Since the underlying graph, all other pipes, and all Q-constraints remain unchanged, $\mathcal{E}'$ is valid embedding of I.

This concludes the proof of correctness of ConvertSmall. As all affected vertices have degree at most 3, the time required to execute the operation is constant. ■

6.3.3 The EncapsulateAndJoin Operation

The purpose of the EncapsulateAndJoin operation is to communicate embedding restrictions between two cut-vertices matched with each other in two steps: First we encapsulate the cut-vertices into their own independent star components, also disconnecting their incident blocks from each other. In the second step, we join the stars. Figures 6.2 and 6.3 show an example.

For an instance $I = (G, \mathcal{P}, \mathcal{Q}, \psi)$ of Synchronized Planarity, let $\rho = (u, v, \varphi_{uv})$ be a pipe matching two cut-vertices u, v of two (not necessarily distinct) connected components C_u, C_v of G. Operation EncapsulateAndJoin(ρ, I) can be applied resulting in an instance $I' = (G', \mathcal{P}', \mathcal{Q}', \psi')$ using the following two steps. We first preprocess both cut-vertices to *encapsulate* them into their own separate star components. Let $C_1, \ldots, C_k$ be the connected components of $C_u - u$. We split C_u along the cuts $(V(C_i), V \setminus V(C_i))$ for $i = 1, \ldots, k$. We denote the vertices resulting from the

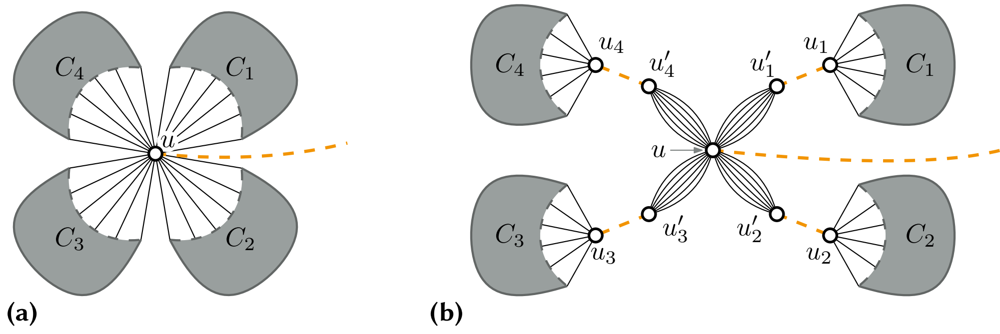


Figure 6.2: A matched cut-vertex **(a)** and the result of encapsulating it **(b)**.

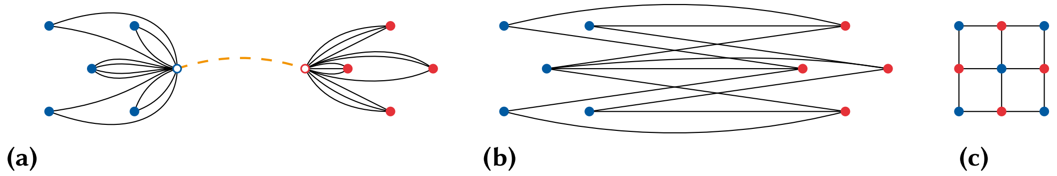


Figure 6.3: Two (encapsulated) matched cut-vertices **(a)**. Depending on the mapping φ, any bipartite graph can result from joining them. For example, the graph in **(b)** can result, which is isomorphic to the square grid graph shown in **(c)**.

split along $(V(C_i), V\setminus V(C_i))$ as u_i and u'_i, where u_i results from contracting $V\setminus V(C_i)$ and u'_i results from contracting $V(C_i)$. Note that, after all splits, u is the center of a star C'_u with ray vertices $u'_1, \ldots, u'_k$. We add the pipes $(u_i, u'_i, \varphi_{u_i u'_i})$ for $i = 1, \ldots, k$; see Figure 6.2. The same procedure is also applied to v, resulting in an intermediate instance I^*. In the second step, we *join* the distinct connected components C'_u and C'_v at u and v along the mapping φ_{uv} of ρ into a component C_{uv}. We also remove the pipe ρ from I^*; all other parts of the instance remain unchanged. Figure 6.3 shows a possible result of joining two stars.

▶ **Lemma 6.4.** Applying EncapsulateAndJoin to a pipe ρ yields an equivalent instance in $O(\deg(\rho))$ time. ◀

Proof. We will first show that the first step yields an equivalent intermediate instance I^*. By Lemma 6.1, a valid embedding of C_u respects each of the cuts $(V(C_i), V \setminus V(C_i))$ for $i = 1, \ldots, k$, yielding a planar embedding for C'_u. The same holds for C_v and C'_v. As all other connected components remain unaffected, we can thus obtain a planar embedding $\mathcal{E}^*$ of I^* from a valid embedding $\mathcal{E}$ of the corresponding instance I. By construction, it is $\mathcal{E}^*(u_i) = \overline{\mathcal{E}^*(u'_i)}$ for $i = 1, \ldots, k$, that is, each new pipe $(u_i, u'_i, \varphi_{u_i u'_i})$ is satisfied and $\mathcal{E}^*$ is a valid embedding of I^*.

Conversely, if $\mathcal{E}^*$ is a valid embedding of I^*, we can join u_i with u'_i for $i = 1, \dots, k$ to obtain a valid planar embedding $\mathcal{E}$ of I, as the pipe $(u_i, u'_i, \varphi_{u_i u'_i})$ ensures that $\mathcal{E}^*$ is compatible with $\varphi_{u_i u'_i}$. The same applies to C_v.

Now consider the instance I' resulting from the second step. If $\mathcal{E}^*$ is a valid embedding for I^*, it satisfies the pipe (u, v, φ_{uv}) and we can join the embedding at u and v via φ_{uv} to obtain a planar embedding $\mathcal{E}'$ of G'. Since the rotations of vertices different from u, v are unaffected, $\mathcal{E}'$ is valid for I'. Conversely, assume that $\mathcal{E}'$ is a valid embedding for I'. Note that joining two stars at their centers yields a bipartite graph consisting of the ray vertices of the former stars. Thus C_{uv} is bipartite, and by Lemma 6.2 every embedding respects the cut of the bipartition. Thus, we can split $\mathcal{E}'$ and obtain a valid embedding of I^*.

As the operation affects exactly the edges incident to u and v and potentially creates a new structure with size proportional to their number, its running time is linear in the degree of the affected pipe. ■

Observe that this operation replaces a pipe by multiple pipes of smaller degrees. Furthermore, all new pipes introduced by the operation have a block-vertex as endpoint. Through multiple applications of `EncapsulateAndJoin` we can thus remove all cut-vertex-to-cut-vertex pipes. Note that `EncapsulateAndJoin` can yield an arbitrary bipartite component. If the component is non-planar, we abort and report a no-instance in our algorithm.

6.3.4 The PropagatePQ Operation

The operation `PropagatePQ` communicates embedding restrictions of a block across a pipe. These restrictions are represented by the embedding tree of the matched P-vertex of interest. Both endpoints of the pipe are replaced by copies of this tree. To ensure that both copies are embedded in a compatible way, we synchronize their inner nodes using pipes and Q-constraints; see Figure 6.4.

For an instance $I = (G, \mathcal{P}, \mathcal{Q}, \psi)$ of Synchronized Planarity, let u be a block-vertex matched by a pipe $\rho = (u, v, \varphi_{uv})$. Operation `PropagatePQ`$(u, I)$ can be applied as follows, resulting in an instance $I' = (G', \mathcal{P}', \mathcal{Q}', \psi')$. We turn the PQ-tree $\mathcal{T}_u$ into a tree T_u by interpreting Q-nodes as Q-vertices and P-nodes as P-vertices. To construct G' from G, we replace u with T_u by reconnecting the incident edges of u to the respective leaves of T_u. We also replace v by a second copy T'_u of T_u by reconnecting an edge e incident to v to the leaf of T'_u that corresponds to $\varphi_{vu}(e)$. For a vertex α of T_u we denote the corresponding vertex of T'_u by α'. For an edge $\alpha\beta$ of T_u we define $\varphi_{T_u T'_u}(\alpha\beta) = \alpha'\beta'$. For each Q-vertex α of T_u, we define $\psi'(\alpha)$ according to the rotation of the corresponding Q-node in $\mathcal{T}_u$. For the Q-vertex α' of T'_u, we define

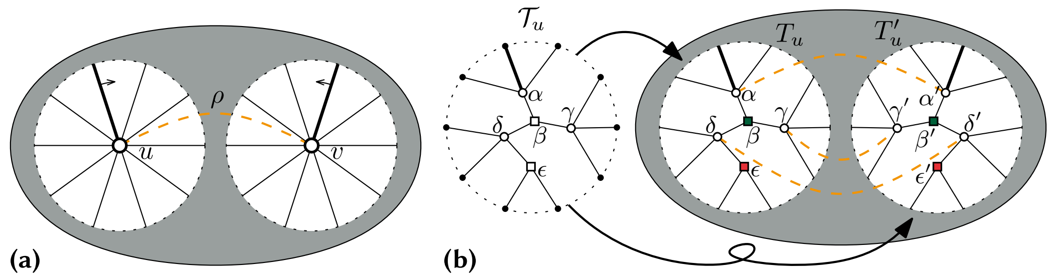


Figure 6.4: A block-vertex u matched with vertex v **(a)**; the bijection φ_{uv} maps the bold edge of u to the bold edge of v, the remaining edges are mapped according to their order, clockwise around u and counter-clockwise around v. The result of applying PropagatePQ(u, I) **(b)**. Note that the second inserted tree T'_u is mirrored with respect to T_u. Q-vertices and -nodes are drawn as squares while P-vertices and -nodes are drawn as disks.

$\psi'(\alpha') = \overline{\varphi_{T_u T'_u}(\psi'(\alpha))}$. For all other Q-vertices of I, ψ' coincides with ψ. We define the partition $\mathcal{Q}' = \mathcal{Q} \cup \{\{\alpha, \alpha'\} \mid \alpha \text{ is a Q-vertex of } T_u\}$. For each P-vertex α of T_u, we define a pipe $\rho_\alpha = (\alpha, \alpha', \varphi_{\alpha\alpha'})$ with $\varphi_{\alpha\alpha'}(e) = \varphi_{T_u T'_u}(e)$ for each edge e incident to α. Finally, we define the matching $\mathcal{P}' = (\mathcal{P} \setminus \{\rho\}) \cup \{\rho_\alpha \mid \alpha \text{ is a P-vertex of } T_u\}$.

▶ **Lemma 6.5.** Applying PropagatePQ to a block-vertex u yields an equivalent instance. If the embedding tree $\mathcal{T}_u$ is known, operation PropagatePQ runs in $O(\deg(u))$ time. ◀

Proof. First we show that PropagatePQ preserves a valid embedding $\mathcal{E}$ of I. To define an embedding $\mathcal{E}'$ for I', we substitute u and v in $\mathcal{E}$ by suitably embedded trees T_u and T'_u. The tree inserted at u represents all the possible rotations of u, including $\mathcal{E}(u)$ and the insertion can therefore be done without introducing crossings. As $\mathcal{E}$ fulfills the P-constraint of the pipe uv, we know that $\mathcal{E}(u) = \overline{\varphi_{vu}(\mathcal{E}(v))}$. Therefore, the same holds for inserting the mirrored copy T'_u of T_u instead of v. Thus, the resulting embedding $\mathcal{E}'$ is planar. Note that the mirror embedding of T_u is obtained by reversing the rotation of each inner vertex of T_u. Therefore, for each inner vertex α of T_u it holds that $\mathcal{E}'(\alpha) = \overline{\varphi_{\alpha'\alpha}(\mathcal{E}'(\alpha'))}$. Thus, all the new pipes are satisfied. Similarly, for each inner Q-vertex α of T_u the rotation in $\mathcal{E}'$ is either $\psi'(\alpha)$ or $\overline{\psi'(\alpha)}$. As the rotation of α' in $\mathcal{E}'$ is mirrored, the new Q-constraints are satisfied. Since all other P- and Q-constraints remain satisfied, $\mathcal{E}'$ is a valid embedding for I'.

Conversely, assume that $\mathcal{E}'$ is a valid embedding for I'. We obtain an embedding $\mathcal{E}$ of I by contracting T_u and T_v into single vertices u and v, respectively. Clearly, $\mathcal{E}$ is a planar embedding. All Q-constraints of I and also all pipes except for ρ remain satisfied. It remains to show that also pipe ρ is satisfied. For each vertex α of T_u

it holds that $\mathcal{E}'(\alpha) = \overline{\varphi_{\alpha'\alpha}(\mathcal{E}'(\alpha'))}$, as $\mathcal{E}'$ in particular fulfills all new Q- and P-constraints of I'. Therefore, the cyclic order of the leaves of T_u is the reverse of the order of the leaves of T'_u. This means that $\mathcal{E}(u) = \overline{\varphi_{vu}(\mathcal{E}(v))}$, that is, ρ is satisfied.

This concludes the proof of correctness of `PropagatePQ`. Note that the operation affects exactly the edges incident to u and v and that the size of an embedding tree is linear in the degree of the represented vertex. Thus, the running time of the operation is linear in the degree of the affected vertices, given that the embedding tree $\mathcal{T}_u$ is known. ■

Note that the tree T'_u inserted instead of v may not be compatible with the rotations of v. In this case, the component becomes non-planar, potentially causing the later generation of an embedding tree to fail. To detect this early, we can also compute the embedding tree $\mathcal{T}_v$ of v and intersect $\mathcal{T}_u$ with $\mathcal{T}_v$ in linear time [Boo75; Pfr20] before the insertion. The effect this has on the practical running time is discussed in Section 9.3.2. Either way, if the generation of an embedding tree or the intersection of two embedding trees fails, we can immediately report a no-instance.

Note that if $\mathcal{T}_u$ is trivial, applying `PropagatePQ` yields an unchanged instance. To make progress in this case, we instead use the operation `SimplifyMatching` described in the following section. Further observe that if $\mathcal{T}_u$ consists of a single Q-node, `PropagatePQ` effectively replaces the affected pipe by two Q-vertices in the same partition cell. Assuming $\mathcal{T}_u$ to be non-trivial, the degrees of all P-vertices in T_u and T'_u are strictly smaller than the degree of u. Thus, by repeatedly applying `PropagatePQ` to vertices with non-trivial embedding tree, we eventually arrive at an equivalent instance where all matched block-vertices have a trivial embedding tree.

6.3.5 The SimplifyMatching Operation

The remaining operation is `SimplifyMatching`, which is used to resolve pipes where one side has no restrictions to be communicated to the other side. This is the case when one of the two matched vertices is a pole of a bond that allows arbitrary rotation. We distinguish three cases: (i) bonds where one pole can always mimic the rotation of the other, (ii) bonds where the pipe synchronizes one pole with the other (similar to the toroidal instances of Fulek and Tóth [FT22]), and (iii) bonds that link two distinct pipes.

For an instance $I = (G, \mathcal{P}, \mathcal{Q}, \psi)$ of Synchronized Planarity, let u be a block-vertex of G whose embedding tree is trivial and that is matched by a pipe ρ. Then, its embedding is determined by exactly one triconnected component μ, which

is a bond.[5] Thus u is the pole of bond μ, and we call the vertex v that is the other pole of μ the *partner* of u. If v is unmatched or a block-vertex with a trivial embedding tree, the operation SimplifyMatching(u, I) can be applied, resulting in an instance $I' = (G', \mathcal{P}', \mathcal{Q}', \psi')$ as follows. Note that, if the embedding tree of v is trivial, the rotation of both vertices is exclusively determined by the embedding of their bond μ. Thus, there are bijections δ_u and δ_v between the edges incident to u and v, respectively, and the virtual edges within the bond. Thereby μ induces a bijection $\delta_{vu} = \delta_u^{-1} \circ \delta_v$ (and conversely $\delta_{uv} = \delta_{vu}^{-1}$) between the edges incident to v and the edges incident to u in this case. Further note that, due to the temporary replacement of Q-vertices by wheels when computing the embedding trees, v cannot be a Q-vertex, as that would make the PQ-tree of u contain a Q-node.

(i) If v is an unmatched P-vertex (Figure 6.5 (i)), I' is obtained from I by removing ρ.

(ii) If ρ matches u with v, it connects the two poles of the bond μ (Figure 6.5 (ii)). Note that, as v is matched, it must have a trivial embedding tree for SimplifyMatching to be applicable. We now check that the bijection δ_{vu} induced by μ is compatible with the bijection φ_{uv} given by the pipe. Let $\pi = \varphi_{uv} \circ \delta_{vu}$ be a permutation of the edges incident to v. If all cycles of π, that is the sequences obtained by repeatedly applying π to an element until reaching the initial element again, have the same length, I' is obtained from I by removing ρ.[6] Otherwise, I is a negative instance and we set I' to a trivial no-instance.

(iii) If v is matched with a P-vertex $v' \neq u$ via pipe $\rho' = (v, v', \varphi_{vv'})$, let u' be the other endpoint of $\rho = (u, u', \varphi_{uu'})$. Again, as v is matched, it must have trivial embedding tree for SimplifyMatching to be applicable. Note that, as u and v are partners with trivial embedding trees, they must have the same degree. We remove ρ and ρ' and add the new pipe $\rho^* = (u', v', \varphi_{u'v'})$ with $\varphi_{u'v'} = \varphi_{vv'} \circ \delta_{uv} \circ \varphi_{u'u}$; see Figure 6.5 (iii).

► **Lemma 6.6.** Applying SimplifyMatching to a block-vertex u with a trivial embedding tree yields an equivalent instance in $O(\deg(u))$ time. ◄

5 This is because a second bond would cause another P-node in the embedding tree, a rigid would cause a Q-node and polygons do not affect the embedding trees [BR16b, Section 2.5].

6 If all cycles of π have the same length, π is *order preserving* and it is $\pi(O) = O$ for any cyclic order O; see [BR16b, Lemma 2.2].

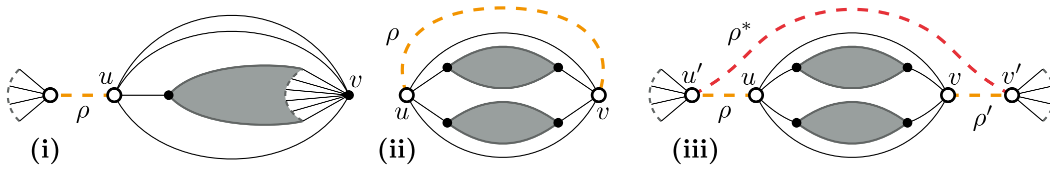


Figure 6.5: The three cases of the SimplifyMatching operation. In case (i) and case (ii), the pipe ρ is removed. In case (iii) the pipes ρ, ρ' are replaced by pipe $\rho^\star$.

Proof. First we show that if $\mathcal{E}$ is a valid embedding of I, then it is also a valid embedding of I'. This clearly holds in case (i), where I' is simply obtained from I by removing a pipe.

It remains to investigate case (ii) and case (iii). In the latter case, let u' be the vertex to which u is matched and let $v' \neq u$ be the vertex to which v is matched. We want to show that $\mathcal{E}$ satisfies the constraint $\mathcal{E}(v') = \overline{\varphi_{u'v'}(\mathcal{E}(u'))}$ of the newly added pipe $\rho^* = (u', v', \varphi_{u'v'})$, where we chose $\varphi_{u'v'} = \varphi_{vv'} \circ \delta_{uv} \circ \varphi_{u'u}$. By assumption the pipes ρ, ρ' are satisfied by $\mathcal{E}$, that is $\mathcal{E}(u) = \overline{\varphi_{u'u}(\mathcal{E}(u'))}$ and $\mathcal{E}(v') = \overline{\varphi_{vv'}(\mathcal{E}(v))}$. Moreover, as both u and v have a trivial embedding tree, the bijections δ_u, δ_v are defined as before. Thus $\delta_u(\mathcal{E}(u))$ and $\delta_v(\mathcal{E}(v))$ define cyclic orders of the virtual edges of μ. Since the embedding is planar, $\delta_u(\mathcal{E}(u)) = \overline{\delta_v(\mathcal{E}(v))}$. This yields $\mathcal{E}(v) = \overline{\delta_{uv}(\mathcal{E}(u))}$ and thus

$$\begin{aligned}\mathcal{E}(v') &= \overline{\varphi_{vv'}(\mathcal{E}(v))} = \overline{\varphi_{vv'}(\overline{\delta_{uv}(\mathcal{E}(u))})} = \varphi_{vv'}(\delta_{uv}(\mathcal{E}(u))) \\ &= \varphi_{vv'}(\delta_{uv}(\overline{\varphi_{u'u}(\mathcal{E}(u'))})) = \overline{\varphi_{vv'} \circ \delta_{uv} \circ \varphi_{u'u}(\mathcal{E}(u'))} = \overline{\varphi_{u'v'}(\mathcal{E}(u'))}\,.\end{aligned}$$

In particular, this means that ρ^* is satisfied and therefore $\mathcal{E}$ is valid.

In case (ii), we consider the permutation $\pi = \varphi_{uv} \circ \delta_{vu}$. Analogously to $\mathcal{E}(v) = \overline{\delta_{uv}(\mathcal{E}(u))}$, it is $\mathcal{E}(u) = \overline{\delta_{vu}(\mathcal{E}(v))}$, and since $\mathcal{E}$ satisfies ρ, we find that $\pi(\mathcal{E}(v)) = \varphi_{uv} \circ \delta_{vu}(\mathcal{E}(v)) = \varphi_{uv}(\overline{\mathcal{E}(u)}) = \mathcal{E}(v)$. All cycles of π have the same length [BR16b, Lemma 2.2] and therefore I' is obtained from I by removing ρ and, in particular, $\mathcal{E}$ is a valid embedding of I'.

Conversely, assume that $\mathcal{E}'$ is a valid embedding for I'. To obtain a valid embedding $\mathcal{E}$ of I, we modify the embedding $\mathcal{E}'$ by changing the order of the virtual edges of the bond μ with poles u and v in such a way that the removed pipes are satisfied. Since, compared to $\mathcal{E}'$, we only change the embedding of a bond, $\mathcal{E}$ is guaranteed to be planar. The details depend on which case of the operation applies.

If v is unmatched in I, we change the embedding of μ such that $\mathcal{E}(u) = \overline{\varphi_{u'u}(\mathcal{E}(u'))}$. This is possible since there is a bijection between the edges incident to u and the virtual edges of μ. The new embedding $\mathcal{E}$ satisfies ρ and, since v is unmatched, also all other pipes of I remain satisfied.

In case (ii), v is matched with u and we consider the above permutation π. As $\mathcal{E}'$ is a valid embedding, I' results from I by removing ρ and we know that all cycles of π have the same length. There exists a cyclic order σ of the edges incident to v with $\pi(\sigma) = \sigma$ [BR16b, Lemma 2.2]. We change the embedding of μ such that $\mathcal{E}(v) = \sigma$ and, to retain planarity, $\mathcal{E}(u) = \overline{\delta_{vu}(\mathcal{E}(v))}$. This satisfies ρ as $\overline{\varphi_{uv}(\mathcal{E}(u))} = \overline{\varphi_{uv}(\overline{\delta_{vu}(\mathcal{E}(v))})} = \varphi_{uv}(\delta_{vu}(\mathcal{E}(v))) = \pi(\mathcal{E}(v)) = \mathcal{E}(v)$ and thus $\mathcal{E}$ is valid.

In case (iii), v is matched with $v' \neq u$. We obtain $\mathcal{E}$ from $\mathcal{E}'$ by setting $\mathcal{E}(u) = \overline{\varphi_{u'u}(\mathcal{E}(u'))}$ and $\mathcal{E}(v) = \overline{\varphi_{v'v}(\mathcal{E}(v'))}$. This satisfies ρ and ρ', and to verify the planarity of $\mathcal{E}$ it suffices to show that $\mathcal{E}(v) = \overline{\delta_{uv}(\mathcal{E}(u))}$. Since ρ^* is satisfied by $\mathcal{E}'$ and $\mathcal{E}$ differs from $\mathcal{E}'$ only at u and v, we know that $\mathcal{E}(v') = \overline{\varphi_{u'v'}(\mathcal{E}(u'))} = \overline{\varphi_{vv'} \circ \delta_{uv} \circ \varphi_{u'u}(\mathcal{E}(u'))}$. This is equivalent to $\varphi_{v'v}(\mathcal{E}(v')) = \overline{\delta_{uv}(\varphi_{u'u}(\mathcal{E}(u')))}$. By the definitions of $\mathcal{E}(u)$ and $\mathcal{E}(v)$, this yields $\mathcal{E}(v) = \overline{\delta_{uv}(\mathcal{E}(u))}$, that is $\mathcal{E}$ is planar.

This concludes the proof of correctness of `SimplifyMatching`. The updates to the matching $\mathcal{P}$ can be done in constant time, while the updates to the bijection φ and the check for cycle lengths require $O(\deg(u))$ time. ■

6.3.6 Reduced and Pipe-Free Instances

With our exposition of the fundamental operations complete, we now study how to solve instances where none of those operations can be applied. We call such instances *reduced*.

▶ **Lemma 6.7.** An instance is reduced if and only if it contains no pipes. ◀

Proof. Obviously, a pipe-free instance is reduced. Conversely, consider a reduced instance I. Assume, for the sake of contradiction, that I contains a pipe. We now show that this implies that one of the operations is applicable, that is, I is not reduced.

First assume that I contains no matched cut-vertices and thus all matched vertices are block-vertices. If there is a matched P-vertex with a non-trivial embedding tree, `PropagatePQ` can be applied. Otherwise, all matched P-vertices are block-vertices with trivial embedding trees and `SimplifyMatching` can be applied. A contradiction.

Second, let u be a matched cut-vertex of maximum degree that is matched to a vertex v by a pipe ρ. If v is also a cut-vertex, we can apply `EncapsulateAndJoin`. If v is a block-vertex with a non-trivial embedding tree, we can apply `PropagatePQ`. Therefore, v must be a block-vertex with a trivial embedding tree. Now we can apply `SimplifyMatching` unless the partner pole v' of v is matched and either a cut-vertex

or a block-vertex with a non-trivial embedding tree. If v' is a matched block-vertex with a non-trivial embedding tree, we can apply `PropagatePQ`. If v' is a matched cut-vertex we have $\deg(u) = \deg(v) < \deg(v')$, contradicting the maximality of $\deg(u)$. The last inequality follows from the fact that $\deg(v) \leq \deg(v')$ already holds in the block of G that contains v and v', but as v' is a cut-vertex, it has at least one neighbor outside that block. ■

To solve instances without pipes in linear time, note that a planar embedding of such an instance is valid if and only if it satisfies the Q-constraints. As Q-vertices only have a binary choice for their rotation, it is relatively easy to synchronize them via a 2-SAT formula. Linear-time algorithms follow from, e.g., [BR16b], and can also be obtained from techniques similar to those used by Fulek and Tóth [FT22] for cubic graphs. For the sake of completeness, we present a self-contained solution.

▶ **Lemma 6.8.** An instance of Synchronized Planarity without pipes can be solved in $O(m)$ time. A valid embedding can be computed in the same time, if it exists. ◀

Proof. We replace each Q-vertex by a wheel of the respective degree. Note that each such wheel is triconnected and entirely contained in a single rigid triconnected component. We now use the decomposition in triconnected components to represent all possible planar embeddings. As the wheel-replacements yield an instance that is linear in the size of the initial instance, this decomposition can be done in $O(m)$ time. If at least one of the rigids has no planar embedding, we abort and report a non-planar instance. It remains to restrict the possible embeddings of the rigids so that all Q-constraints are satisfied.

To do so, we construct an instance of 2-SAT, where each solution corresponds to a planar embedding $\mathcal{E}$ that is a valid solution for I. For every Q-vertex v the Boolean variable x_v is `true` if the rotation of v in $\mathcal{E}$ is equal to the default rotation of v (i.e. if $\mathcal{E}(v) = \psi(v)$) and `false` otherwise (i.e. if $\mathcal{E}(v) = \overline{\psi(v)}$). Additionally, for every Q-constraint cell $Q \in \mathcal{Q}$ we add a boolean variable x_Q, and for every Q-vertex $v \in Q$ we add the constraint $(x_v \vee \neg x_Q) \wedge (\neg x_v \vee x_Q)$. This ensures that the rotations of the Q-vertices are consistent within their cell and thus satisfy the Q-constraints. We still need to ensure that the 2-SAT instance allows only planar embeddings. For each rigid μ, we fix one of its two planar embeddings as its default embedding $\mathcal{E}_\mu$ and add another boolean variable x_μ, indicating whether $\mathcal{E}_\mu$ or $\overline{\mathcal{E}_\mu}$ shall be used in $\mathcal{E}$. Due to the wheel replacement, each Q-vertex v is entirely contained in its rigid μ, which can be found in constant time using one of the incident edges. For every Q-vertex v we now add one of the following two constraints with regard to its

rigid μ: either 1) $(x_v \vee \neg x_\mu) \wedge (\neg x_v \vee x_\mu)$ if $\psi(v) = \mathcal{E}_\mu(v)$, or 2) $(x_v \vee x_\mu) \wedge (\neg x_v \vee \neg x_\mu)$ if $\psi(v) = \overline{\mathcal{E}_\mu(v)}$. This ensures that the rotation of every Q-vertex is consistent with the planar embedding of its rigid.

In the resulting 2-SAT instance, we have a Boolean variable for each Q-vertex, Q-constraint and rigid, and four constraints for each Q-vertex. The constructed 2-SAT formula thus has size $O(m)$ and can be solved in linear time [APT79]. If it has no solution, we report an invalid instance and abort. Otherwise, we can use x_μ to decide whether $\mathcal{E}_\mu$ should be mirrored or not. Choosing any planar embedding for each bond, i.e. a permutation of the parallel virtual edges between the two poles, this yields a planar embedding $\mathcal{E}$ that is a valid solution for I. ■

6.3.7 Finding a Reduced Instance

As mentioned above, we exhaustively apply the operations EncapsulateAndJoin, PropagatePQ, and SimplifyMatching. We claim that this algorithm terminates and yields a reduced instance after a linear number of steps. The key idea is that the operations always make progress by either reducing the number of pipes, or by splitting pipes into pipes of smaller degree. This suggests that, eventually, we arrive at an instance without pipes. However, there are two caveats. First, the encapsulation in the first step of EncapsulateAndJoin creates new pipes and thus has the potential to undo progress. Second, the smaller pipes resulting from splitting a pipe with PropagatePQ might cause further growth of the instance, potentially causing a super-linear number of steps.

We resolve both issues by using a more fine-grained measure of progress in the form of a potential function. To overcome the first issue, we show that for each application of EncapsulateAndJoin, the progress that is undone in the first step is outweighed by the progress made through the following join in the second step. Similarly, for the second issue, we show that the sum of the parts is no bigger than the whole when splitting pipes.

As P-vertices of degree 3 or less are converted to Q-vertices (see Section 6.3.2), we use $\deg^*(u) = \deg^*(v) = \deg^*(\rho) = \max\{\deg(x) - 3, 0\}$ to denote the number of incident edges that keep a P-vertex u (and also the other endpoint v of its pipe $\rho = (u, v, \varphi_{uv})$) from becoming converted to a Q-vertex. We also partition the set of all pipes $\mathcal{P}$ into the two cells $\mathcal{P}_{CC}$ and $\mathcal{P}_B = \mathcal{P} \setminus \mathcal{P}_{CC}$, where $\mathcal{P}_{CC}$ contains all pipes where both endpoints are cut-vertices. We define the *potential* of an instance I as $\Phi(I) = \sum_{\rho \in \mathcal{P}_B} \deg^*(\rho) + \sum_{\rho \in \mathcal{P}_{CC}} (2 \deg^*(\rho) - 1)$.

We show that the operations always decrease this potential. To analyze the potential change of PropagatePQ and EncapsulateAndJoin, we need the following technical lemma for bounding the sum of the degrees of multiple smaller pipes replacing a single bigger pipe.

► **Lemma 6.9.** Let $k \geq 2$, $d_1 \geq d_2 \geq \ldots \geq d_k \geq 1$ and $c \geq 0$ be integers. Let $j = |\{i \mid d_i \geq 3\}|$ and let $\ell = |\{i \mid d_i = 2\}|$. If $3 \leq c + \ell + 2j$, then $k + \sum_{i=1}^{k} \max\{d_i - 3, 0\} \leq c - 3 + \sum_{i=1}^{k} d_i$. If $4 \leq \sum_{i=1}^{k} d_i$, then $\sum_{i=1}^{k} \max\{d_i - 3, 0\} \leq -4 + \sum_{i=1}^{k} d_i$. ◄

Proof. Observe that the d_i are ordered non-increasing and thus $d_i \geq 3$ for $i = 1, \ldots, j$ and $d_i < 3$ for $i = j+1, \ldots, k$. More specifically, it is $d_{j+1}, \ldots, d_{j+\ell} = 2$ and $d_{j+\ell+1}, \ldots, d_k = 1$. This yields $\sum_{i=j+1}^{j+\ell} d_i = 2 \cdot \ell$ and $\sum_{i=j+\ell+1}^{k} d_i = k - \ell - j$ and we can also avoid the "max" using $\sum_{i=1}^{k} \max\{d_i - 3, 0\} = \sum_{i=1}^{j}(d_i - 3) = -3j + \sum_{i=1}^{j} d_i$. We now start at $3 \leq c + \ell + 2j$, which can be rewritten as $k - 3j \leq c - 3 + 2\ell + (k - \ell - j)$. Adding $\sum_{i=1}^{j} d_i$ on both sides and using the above observations yields

$$k + \sum_{i=1}^{k} \max\{d_i - 3, 0\} = k - 3j + \sum_{i=1}^{j} d_i \leq c - 3 + \sum_{i=j+1}^{j+\ell} d_i + \sum_{i=j+\ell+1}^{k} d_i + \sum_{i=1}^{j} d_i = c - 3 + \sum_{i=1}^{k} d_i.$$

Note that inserting $c = k - 1$ in the first formula yields the second formula. It remains to show that in this case $3 \leq c + \ell + 2j$ or the equivalent $4 \leq k + \ell + 2j$ follow from $4 \leq \sum_{i=1}^{k} d_i$. If $k \geq 4$, the inequality obviously always holds. If $k = 3$, it must be $j \geq 1$ or $\ell \geq 1$ as the sum is at least 4. If $k = 2$, it must be $j \geq 1$ or $\ell \geq 2$ as the sum is at least 4. ■

► **Lemma 6.10.** For an instance $I = (G, \mathcal{P}, \mathcal{Q}, \psi)$ of Synchronized Planarity and an instance $I' = (G', \mathcal{P}', \mathcal{Q}', \psi')$ that results from application of either EncapsulateAndJoin, PropagatePQ or SimplifyMatching to I, the following three properties hold:

(i) The potential reduction $\Delta\Phi = \Phi(I) - \Phi(I')$ is at least 1.

(ii) The number of nodes added to the graph satisfies $\Delta V = |V(G')| - |V(G)| \leq 2 \cdot \Delta\Phi + 12$.

(iii) If the operation replaces a connected component C by one or multiple connected components, then each such component C' satisfies $\Delta E(C) = |E(C')| - |E(C)| \leq 2 \cdot \Delta\Phi$. ◄

Proof. We analyze the effects of EncapsulateAndJoin, PropagatePQ, and SimplifyMatching on the measures $\Delta\Phi$, ΔV and $\Delta E(C)$ and show that the found changes satisfy the claimed bounds.

Operation EncapsulateAndJoin(ρ, I) in the first step encapsulates both cut-vertices u, v into their own star components. For each block incident to u, this introduces two new vertices that are connected by a new pipe. Let $d_1 \geq d_2 \geq \ldots \geq$

$d_{k-1} \geq d_k \geq 1$ be the degrees of the $k \geq 2$ ray vertices of u after the encapsulation. As one end of the added pipes is a block-vertex, the potential is increased by $\sum_{i=1}^{k} \max\{d_i - 3, 0\}$. The ray vertices around v increase the potential and number of vertices likewise, where $d'_1 \geq d'_2 \geq \ldots \geq d'_{k'-1} \geq d'_{k'} \geq 1$ are the degrees of the $k' \geq 2$ ray vertices of v after the encapsulation. Using $\sum_{i=1}^{k} d_i = \sum_{i=1}^{k'} d'_i = D$ it is $\deg(\rho) = \deg(v) = \deg(u) = D$ and $\deg^*(\rho) = \deg^*(v) = \deg^*(u) = \max\{D - 3, 0\} = D - 3$ as u and v are P-vertices of the same degree greater then three. In the second step, since ρ connects two cut-vertices, removing ρ together with its endpoints reduces the potential by $2\deg^*(\rho) - 1$ and we thus get $\Delta\Phi = 2 \cdot (D-3) - 1 - \sum_{i=1}^{k} \max\{d_i - 3, 0\} - \sum_{i=1}^{k'} \max\{d'_i - 3, 0\}$.

As $D \geq 4$, we know from the second formula of Lemma 6.9 that $\sum_{i=1}^{k} \max\{d_i - 3, 0\} \leq (\sum_{i=1}^{k} d_i) - 4 = D - 4$ and also $\sum_{i=1}^{k'} \max\{d'_i - 3, 0\} \leq D - 4$. Using this inequality in the formula above yields $\Delta\Phi = 2D - 7 - \sum_{i=1}^{k} \max\{d_i - 3, 0\} - \sum_{i=1}^{k'} \max\{d'_i - 3, 0\} \geq 2D - 7 - (D-4) - (D-4) = 1$ as claimed by (i).

As the encapsulation generates two vertices for each block and the join removes two vertices, we have $\Delta V = 2k + 2k' - 2$. Lemma 6.9 with $c = 3$ yields $k \leq D - \sum_{i=1}^{k} \max\{d_i - 3, 0\}$ and $k' \leq D - \sum_{i=1}^{k'} \max\{d'_i - 3, 0\}$ independently from the values of ℓ and j. Now claim (ii) holds as

$$\Delta V \leq 2 \cdot \left(D - \sum_{i=1}^{k} \max\{d_i - 3, 0\} + D - \sum_{i=1}^{k'} \max\{d'_i - 3, 0\} - 1 \right) = 2 \cdot (\Delta\Phi + 6).$$

In the first step of EncapsulateAndJoin, two new components with $\deg(u) = \deg(v) = D$ edges each are added, which are then pairwise combined in the second step. This yields a new component with D edges total, which is no bigger than the prior components of u or v (which contained at least the D edges incident to u or v). Thus, claim (iii) holds.[7]

Operation PropagatePQ(u, I) replaces the pipe ρ having block-vertex u as one endpoint by one pipe for each inner P-node of the non-trivial embedding tree $\mathcal{T}_u$ of u. If $\mathcal{T}_u$ only consists of a single Q-node, we are removing a pipe without adding any new pipes, nodes, or edges, thus properties (i)–(iii) are trivially satisfied. Otherwise, let $d_1 \geq d_2 \geq \ldots \geq d_{k-1} \geq d_k$ be the degrees of the $k \geq 2$ inner vertices of $\mathcal{T}_u$. The tree $\mathcal{T}_u$ has $\deg(u)$ leaves and thus $(\sum_{i=1}^{k} d_i) + \deg(u) = 2 \cdot |E(\mathcal{T}_u)| = 2 \cdot (|V(\mathcal{T}_u)| - 1) = 2 \cdot (k + \deg(u) - 1)$ or, equivalently, $\deg(u) = \left(\sum_{i=1}^{k} (d_i - 2)\right) + 2$.

7 Note that due to Lemma 6.1, we could also apply the Join part of the EncapsulateAndJoin operation instead PropagatePQ on pipes where both endpoints are block-vertices. At this point, we can see that this modified approach would violate claim (iii), breaking our running time analysis. We also analyze the effect this has on the practical running time in Section 9.3.2.

As u is a P-vertex with degree at least 4, $\deg^*(u) = \deg(u) - 3 = \left(\sum_{i=1}^{k}(d_i - 2)\right) - 1$. As $\mathcal{T}_u$ contains no vertices of degree 2, it is $d_i \geq 3$ for $i = 1, \ldots, k$. As the added pipes all have one endpoint in the biconnected component of u, the potential is reduced by $\Delta\Phi = \deg^*(u) - \sum_{i=1}^{k} \max\{d_i - 3, 0\} = \left(\sum_{i=1}^{k}(d_i - 2)\right) - 1 - \left(\sum_{i=1}^{k}(d_i - 3)\right) = k - 1 \geq 1$, which shows (i).

Moreover, we replace the two endpoints of ρ each by the inner nodes of $\mathcal{T}_u$, yielding $\Delta V = 2k - 2$ additional nodes. Note that $\Delta V = 2k - 2 = 2 \cdot \Delta\Phi < 2 \cdot \Delta\Phi + 12$ as claimed by (ii).

As each inner node except for the root has one edge connecting it to its parent, we also add $\Delta E(C) = k - 1$ additional edges to each component. Observe that $\Delta E(C) = k - 1 < 2k - 2 = 2 \cdot \Delta\Phi$ as claimed by (iii).

Operation SimplifyMatching always removes at least one pipe $\rho \in \mathcal{P}_B$ and thus decreases the potential by at least $\deg^*(\rho)$. If two pipes ρ, ρ' are replaced by their transitive shortcut (i.e. case (iii) of SimplifyMatching applies), this adds a new pipe ρ^*. If at least one endpoint of ρ^* is a block-vertex, the potential change is $\Delta\Phi = 2\deg^*(\rho) - \deg^*(\rho)$. Otherwise, both endpoints are cut-vertices and ρ^* belongs to $\mathcal{P}_{CC}$, yielding a potential change of $\Delta\Phi = 2\deg^*(\rho) - (2\deg^*(\rho) - 1) = 1$. As no vertices or edges are added or removed $\Delta V = \Delta E(C) = 0$, which is less than $\Delta\Phi$. ■

With this lemma, we know that each step decreases the potential by at least 1 without growing the graph too much. The following shows an upper bound on the potential.

▶ **Lemma 6.11.** For Synchronized Planarity instance I we have $\Phi(I) < 2m$. ◀

Proof. Each pipe ρ matching two vertices u and v contributes at most $2\deg^*(\rho) - 1 < \deg(u) + \deg(v)$ to the potential. Since each vertex is part of at most one pipe, the sum of all potentials is bounded by $\sum_{v \in V} \deg(v) = 2m$. ■

This can be used to bound the size of instances resulting from applying multiple operations consecutively and finally to bound the time required to find a solution for an instance.

▶ **Theorem 6.12.** Synchronized Planarity can be solved in $O(m^2)$ time. ◀

Proof. By Lemma 6.10 the potential function decreases with each applied operation. Therefore, by Lemma 6.11, after $k \leq 2m$ operations a reduced instance $I' = (G', \mathcal{P}', \mathcal{Q}', \psi')$ is reached. We claim that the resulting graph $G' = (V', E')$ has $|V'| \leq |V| + 4 \cdot |E| + 12 \cdot k$ vertices and each connected component C' of G' has $|E(C')| \leq 5 \cdot |E|$ edges.

Let $\Delta\Phi_i$ for $i \in \{1, \dots, k\}$ be the potential reduction caused by the ith applied operation. According to Lemma 6.10, this operation also added $\Delta V_i \leq 2 \cdot \Delta\Phi_i + 12$ vertices to the graph. By Lemma 6.11 we have $\sum_{i=1}^{k} \Delta\Phi_i \leq \Phi(I) < 2|E|$ and thus $|V'| = |V| + \sum_{i=1}^{k} \Delta V_i \leq |V| + 4 \cdot |E| + 12 \cdot k$. Additionally, if the ith operation replaces a connected component C_{i-1} by one or multiple connected components, then each such component C_i satisfies $|E(C_i)| - |E(C_{i-1})| \leq 2 \cdot \Delta\Phi_i$. Each connected component C_1 of the initial graph G has at most $|E|$ edges. Using the same argument as above, we obtain $|E(C')| \leq |E| + 4 \cdot |E|$.

As $m \in \Omega(n)$, this shows that the resulting instance has $O(m)$ vertices and each connected component has $O(m)$ edges. Computing the embedding trees for a single connected component on demand can thus be done in $O(m)$ time. In addition to this computation, each of the k operations takes time linear in the degree of the vertex it is applied to, which also is in $O(m)$. Thus, each operation takes $O(m)$ time and, in total, it takes $O(m^2)$ time to reach a reduced instance. As the size of this reduced instance is also in $O(m^2)$, using Lemma 6.8 for finding a solution for the reduced instance can be done in $O(m^2)$ time. ■

6.3.8 Generating embedding trees efficiently

A major bottleneck for the running time is that embedding trees need to be obtained each time before `PropagatePQ` or `SimplifyMatching` can be applied, which takes time linear in the size of the affected component by first performing a planarity test (see Section 4.2) or computing an SPQR-tree (see Section 7.4.1 and [BR16b, Section 2.5]). We note that, using the fully-dynamic data structure by Eppstein et al. [Epp+98], the SPQR-tree can also be maintained throughout all operations instead of recomputing it each time. Observe that `SimplifyMatching` does not change the graph structure and `EncapsulateAndJoin` does not modify edges within biconnected components, but only generates new, small components whose SPQR-trees can also be computed efficiently. Thus, the only operation that invalidates a pre-computed SPQR-tree is `PropagatePQ`. Using the dynamic SPQR-tree, the expansion of vertices into (embedding) trees used in `PropagatePQ` can be implemented by a sequence of edge deletions as well as vertex and edge insertions, each happening in amortized $O(\sqrt{m})$ time [Epp+98]. Given the SPQR-tree, the embedding tree of a vertex can be computed in time linear in its degree [BR16b]. Thus, this reduces the time needed to apply an operation from $O(m)$ – dominated by the embedding tree computation – to linear in the maximum pipe degree Δ for the operation itself plus the square-root factor for the SPQR-tree updates, that is $O(\Delta \cdot \sqrt{m})$. The overall running time of our algorithm can thereby be reduced to $O(m \cdot \Delta \cdot \sqrt{m})$. Note that unfortunately the recent improvements by Holm and

Rotenberg are not applicable here, as they maintain triconnectivity in an only incremental setting [HR20b], while maintaining only planarity information in the fully-dynamic setting [HR20a]. In Chapter 7 we will describe a version of the SPQR-tree that allows for efficient dynamic updates that cover all changes made by our operations. This allows us to further reduce the running time of each individual operation to $O(\Delta)$ and the running time of the whole Synchronized Planarity algorithm to $O(m \cdot \Delta)$.

6.4 Applications

In this section we discuss several problems that can be solved efficiently by reducing them to Synchronized Planarity. For constrained planarity problems, we can often assume the input to be simple and planar and thus the number of edges m is linear in the number of vertices n. In these cases we give the running time depending on n instead of m.

First, in Sections 6.4.1 and 6.4.2, we will show how Synchronized Planarity can be used to solve the Atomic Embeddability and Clustered Planarity problems. In Sections 6.4.3 to 6.4.5 we consider problems further up in the hierarchy, namely Partially Embedded and Level Planarity as well as variations thereof. A reduction for Connected SEFE-2 is given in Section 6.4.6. The following Sections 6.4.7 and 6.4.8 treat constraints to rotations in the form of PQ-trees. A reduction for a variation of the NodeTrix Planarity Problem is given in Section 6.4.9. Finally, Section 6.4.10 shows that Synchronized Planarity can also be extended to handle pipes where the rotations of the endpoints need to be exactly the same and not the reverse of each other. We apply this to solve restricted instances of a constrained planarity variant where multiple PQ-trees hierarchically constrain the rotation of individual vertices.

6.4.1 Atomic Embeddability

Recall from Section 6.2 that Atomic Embeddability has two graphs as input. One graph represents a molecule structure with atoms and pipes between them, the other graph is mapped onto that structure such that edges connect vertices on a single atom or vertices on neighboring atoms through the corresponding pipe.

▶ **Theorem 6.13.** Atomic Embeddability can be solved in $O(m^2)$ time. ◀

Proof. As observed by Fulek and Tóth [FT22, Observation 1], Atomic Embeddability can be equivalently viewed as follows. For each atom consider the graph on that atom together with, for each incident pipe, one virtual vertex that is incident to all

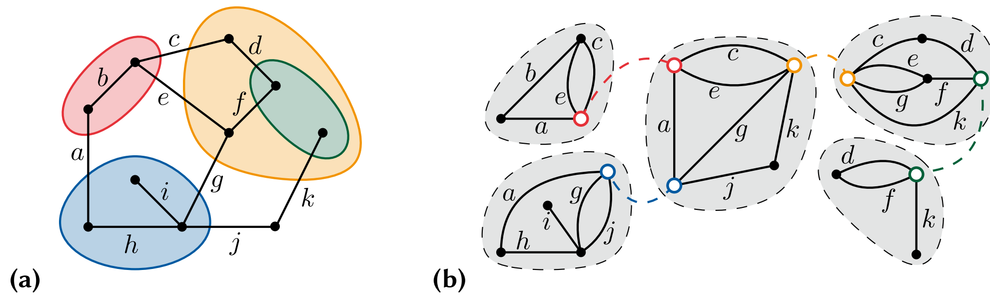


Figure 6.6: An instance of Clustered Planarity **(a)** and its CD-tree representation **(b)**, where each skeleton is shown with a gray background and the virtual vertices are shown as colored disks.

edges that would normally go through this pipe to a neighboring atom. Note that each pipe has two virtual vertices corresponding to it, one on each of its incident atoms. Then an instance of Atomic Embeddability is positive if and only if all these graphs can be embedded in the plane such that every pair of virtual vertices corresponding to the same pipe have opposite rotation. This directly reduces Atomic Embeddability to Synchronized Planarity. ■

6.4.2 Clustered Planarity

To reduce Clustered Planarity to Synchronized Planarity, we use the CD-tree [BR16a]; see also Figure 6.6. Each node of the CD-tree corresponds to a graph, called its *skeleton*. Some vertices of a skeleton are *virtual vertices*. Each virtual vertex corresponds to exactly one virtual vertex in a different skeleton, called its *twin*, and there is a bijection between the edges incident to a virtual vertex and its twin. The tree structure of the CD-tree comes from these correspondences between twins, that is, the CD-tree has an edge between two nodes if and only if their skeletons have virtual vertices that are twins of each other. It is known that a clustered graph is c-planar if and only if the skeletons of all nodes in its CD-tree can be embedded such that every virtual vertex and its twin have opposite rotation [BR16a, Theorem 1].[8] As the CD-tree has linear size and can be computed in linear time, this yields a linear reduction from Clustered Planarity to Synchronized Planarity.

▶ **Theorem 6.14.** Clustered Planarity can be solved in $O((n+d)^2)$ time, where d is the number of crossings between an edge and a cluster boundary. ◀

8 The theorem originally requires "the same" instead of "opposite" rotations. As the CD-tree is acyclic, the embedding of a nested cluster can easily be mirrored without affecting any other parts of the instance, and both formulations can be seen as equivalent.

Proof. We use the disjoint union of all skeletons of the CD-tree and match each virtual vertex with its twin using a pipe. We can assume that the underlying graph of a Clustered Planarity instance has no multi-edges and it must be planar to be cluster-planar, thus its number of edges satisfies $m \in O(n + d)$ and the running time of our algorithm is $O((n + d)^2)$. ■

Interestingly, the reduction from Clustered Planarity to Synchronized Planarity to some degree also works in the other direction: Consider the graph obtained by contracting each connected component of a Synchronized Planarity instance into a single vertex and interpreting its pipes as edges between the contracted vertices. If this graph is simple and acyclic, we can interpret it as the CD-tree of an instance of Clustered Planarity, using the contracted connected components as skeletons where pipe endpoints are interpreted as virtual vertices. Unfortunately, this duality does not necessarily hold for all instances generated during a run of our Synchronized Planarity algorithm. Once an application of PropagatePQ inserts a PQ-tree with two P-nodes, we get two parallel pipes between the same components and thus an illegal multi-edge in the CD-tree. Moreover, this multi-edge might develop into a cycle of pipes requiring an application of the toroidal case of SimplifyMatching, as shown in Figure 6.7. This makes it improbable that our operations could also be directly applied to the Clustered Planarity instance without reducing to Synchronized Planarity first.

6.4.3 Partially Embedded Planarity

Schaefer [Sch13] provides a reduction of Partially Embedded Planarity to SEFE-2, where the shared graph corresponds to the preembeded graph. One exclusive graph additionally contains the unembedded edges, while the other triangulates the preembeded graph to fix its embedding. The generated SEFE-2 instance has a connected shared graph if the preembeded graph is connected, thus only allowing some of the instances obtained via the reduction to be solved by Synchronized Planarity. To use Synchronized Planarity for all Partially Embedded Planarity instances, we provide an alternative reduction to Clustered Planarity. For an instance $(G, H, \mathcal{H})$ of Partially Embedded Planarity, let $\delta_H = |E(H)| \cdot \Delta(H)$ where $\Delta(H)$ is the maximum degree of a vertex in H.

▶ **Theorem 6.15.** Partially Embedded Planarity can be reduced to Clustered Planarity in time $O(|G| + \delta_H)$. The resulting number of clusters is in $O(|H|)$ and the total number of cluster-boundary edge crossings is in $O(|G| + \delta_H)$. ◀

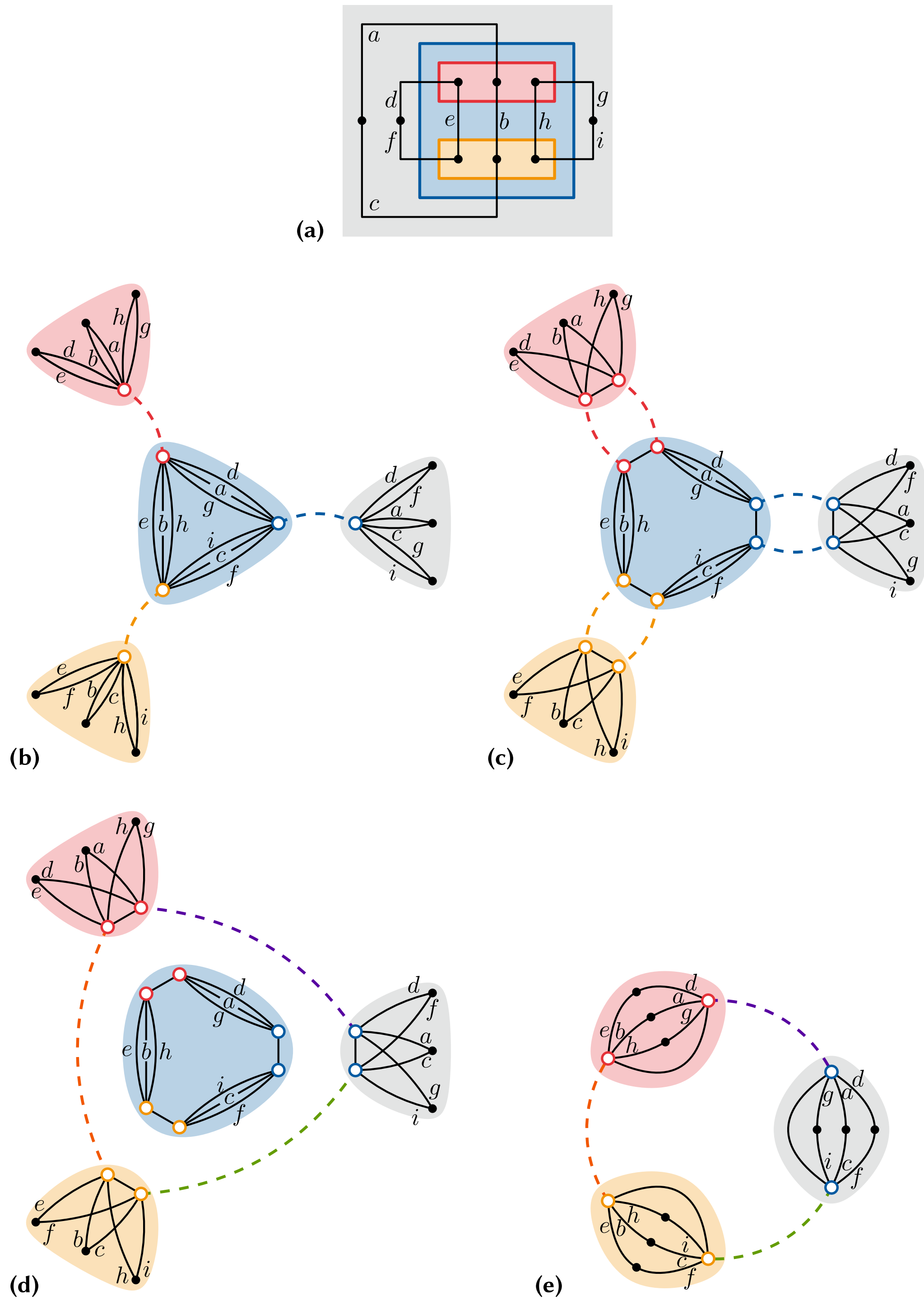


Figure 6.7: **(a)** An instance of Clustered Planarity and **(b)** its CD-tree representation, which can also be interpreted as instance of Synchronized Planarity. **(c)** The result of applying PropagatePQ to each of the three pipes of this instance. **(d)** The result of furthermore applying the transitive case of SimplifyMatching three times. **(e)** The same instance as in (d) with a different layout, ignoring the middle component. Another two applications of the transitive case of SimplifyMatching, followed by an application of the toroidal case make the instance pipe-free.

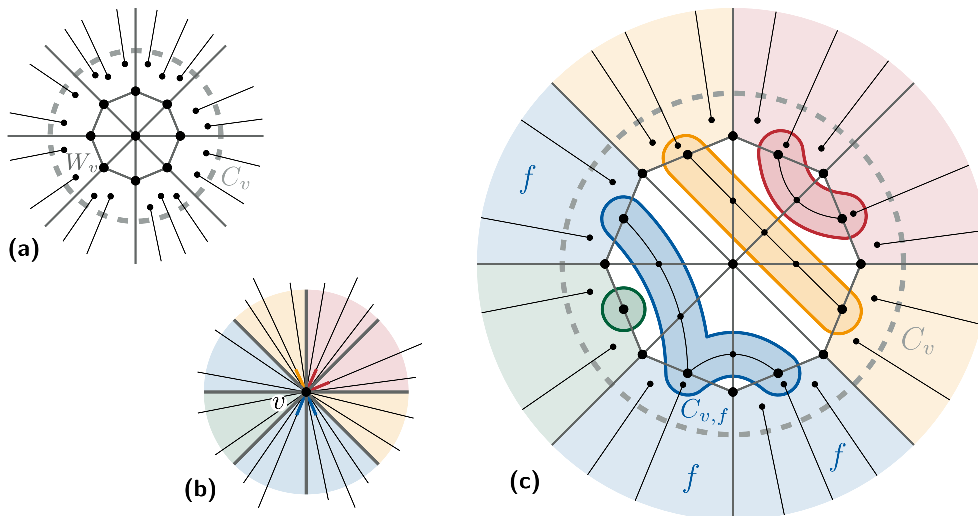


Figure 6.8: (a) The wheel W_v and cluster C_v (dashed) replacing a vertex v in the reduction from PARTIALLY EMBEDDED PLANARITY to CLUSTERED PLANARITY. **(b)** Vertex v before the replacement. Bold gray edges represent edges from H whose embedding is fixed. The four incident faces of $\mathcal{H}$ are shaded in different colors. Edges with a colored tip are the beginning of a path that needs to be embedded in a face of the same color. **(c)** The clusters used to represent the faces of $\mathcal{H}$ around v. The blue cluster $C_{v,f}$ restricts two edges to lie within face f, which is incident three times.

Proof. Let $(G, H, \mathcal{H})$ be an instance of Partially Embedded Planarity, for which we will construct an equivalent instance $I = (G', T)$ of Clustered Planarity. Angelini et al. [Ang+15b, Theorem 4.14] show that we can assume G to be connected without loss of generality. We convert each vertex v of H into a wheel W_v using its embedding from $\mathcal{H}$, adding all wheels and the edges of H between the wheels to G'; see Figure 6.8 (a). The vertices of wheel W_v are added to a new cluster $C_v \in T$. Furthermore, we add all edges and vertices that are exclusive to G to G', replacing any edge endpoint v that is in H with a degree-1 vertex $u \in C_v$. Note that contracting all clusters in a solution of this instance I yields a plane graph that is isomorphic to G and whose embedding respects the rotations of $\mathcal{H}$.

It remains to ensure the correct relative positions of the connected components of $\mathcal{H}$ [Ang+15b, Lemma 2.1]. As explained in Section 5.1, it suffices to ensure that the edges $E(G) \setminus E(H)$ are embedded in the right faces of $\mathcal{H}$. We will once again use clusters to ensure this. For each wheel W_v replacing a vertex v, we subdivide the edges on the outside of the wheel. Each subdivision vertex u incident to face f of $\mathcal{H}$ incident to v is added to a cluster $C_{v,f} \subset C_v$; see Figure 6.8 (c). Note that faces appearing multiple times around a single vertex would now cause the instance to turn non-clustered-planar. To remedy this, we connect all subdivision vertices belonging to the same face by adding edges within the respective cluster in a clockwise order to the inside of the wheel, planarizing and adding to the respective cluster any crossings with the wheel produced by this. As faces appearing multiple times around a single vertex must be non-alternating (they may only be nested within each other), inserting the edges following the nesting from inside to outside ensures that the added edges do not cross each other but only parts of the wheel. Finally, any degree-1 vertex u replacing the endpoint $v \in H$ of an edge $e \in E(G) \setminus E(H)$ that must be embedded in face f is added to the cluster $C_{v,f}$.

A Partially Embedded Planarity solution can easily be converted to a solution of the corresponding Clustered Planarity instance by performing the conversion while maintaining the embedding. Observe that all created clusters are connected when ignoring the degree-1 endpoints of edges from $E(G) \setminus E(H)$, which makes it easy to enclose the connected vertices in a region. For an edge $e \in E(G) \setminus E(H)$ with an original endpoint $v \in H$ that was replaced with a degree-1 vertex v', we can place v' sufficiently close to the wheel W_v replacing v to be within the region of C_v. If e is restricted to be embedded in a face f, all sides of the wheel incident to f have a subdivision vertex that is in $C_{v,f}$ by construction. As we have a valid embedding, e is contained in f and we can place v' sufficiently close to such subdivision vertex such that it lies within the region $C_{v,f}$. Conversely, a Clustered Planarity solution can be converted to a Partially Embedded Planarity solution by contracting all clusters replacing vertices of H back into single vertices. As the resulting embedding

clearly respects the rotations of $\mathcal{H}$, we focus on the relative positions, that is, on the edges that have a prescribed face they need to be embedded in. Note that, for each such edge originally connected to a vertex $v \in H$, only one endpoint v' lies within C_v. This especially means that v' has to lie outside of W_v (but within the region of C_v). Additionally, if e is restricted to lie within face f, v' has to lie within the region of $C_{v,f}$. As it cannot lie within W_v, it follows that v' shares a face with a vertex x that subdivides an outer edge of wheel W_v and also lies within $C_{v,f}$. Contracting v' into x first, we see that e will be embedded within f when contracting C_v, satisfying its constraint. This can be done for each such edge, which shows that the obtained Clustered Planarity instances also ensures correct relative positions.

The planarization adds $O(\delta_H)$ vertices of degree 4 as we have $O(\Delta(H))$ such vertices subdividing any edge of G. The planarization can be performed in $O(\delta_H)$ time, while the remaining reduction runs in time linear in G. This procedure generates one cluster for each vertex of H plus at most one cluster for each edge of H (or more specifically, for each face of $\mathcal{H}$) incident to a vertex of H. Each edge $e = uv$ of G crosses at most 4 cluster boundaries, namely those of the clusters C_v, C_u as well as $C_{v,f}$ and $C_{u,f}$ if it needs to be embedded in face f. The wheels together with their edges subdivided by the planarization contribute at most $O(\delta_H)$ additional crossings. ■

6.4.4 (Radial) Level and Strip Planarity

Schaefer [Sch13] shows that Radial Level Planarity (and thereby also Level Planarity) reduces to Clustered Planarity using a construction using one cluster per vertex. As this creates many crossings between edges and cluster boundaries, which are a main factor in the practical running time of Synchronized Planarity, we provide an alternative reduction that uses far less clusters.

▶ **Theorem 6.16.** Proper Radial Level Planarity can be reduced to Clustered Planarity in linear time, with the number of clusters linear in the number of levels and the number of crossings between edges and cluster boundaries linear in the number of vertices. ◀

Proof. Let (G, γ) be an instance of proper Radial Level Planarity, for which we will construct an equivalent instance $I = (G', T)$ of Clustered Planarity. First, we split each level L_i into two levels L_i^u, L_i^l as follows; see also Figure 6.9. Each vertex $v \in L_i$ is split into two halves v^u and v^l, connected by an edge. The edges of v leading to a higher level are assigned to v^u, while those leading to a lower level are

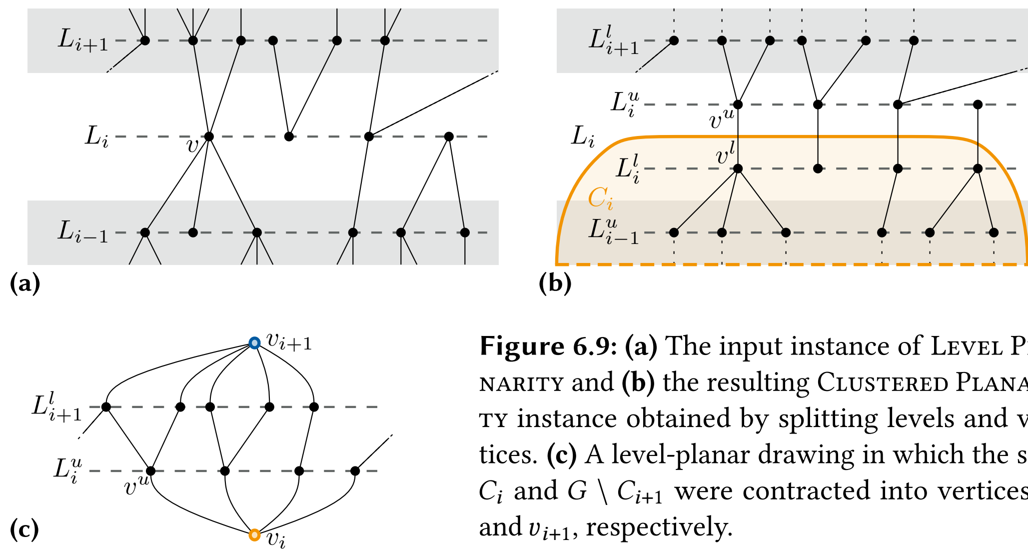


Figure 6.9: (a) The input instance of Level Planarity and **(b)** the resulting Clustered Planarity instance obtained by splitting levels and vertices. **(c)** A level-planar drawing in which the sets C_i and $G \setminus C_{i+1}$ were contracted into vertices v_i and v_{i+1}, respectively.

assigned to v^l. As the new edges ensure the order of vertices on L^u_i equals the order of the corresponding vertices on L^l_i, this yields an equivalent instance of Radial Level Planarity.

We now convert this to an instance of Clustered Planarity by, for each level $i = 1, \ldots, k+1$, interpreting the levels $L^l_1, \ldots, L^l_i$ together with $L^u_1, \ldots, L^u_{i-1}$ as cluster C_i. For $i = 1, \ldots, k$, C_i is a child of C_{i+1}. A drawing that is a solution to the Radial Level Planarity instance can easily be converted to a solution of the Clustered Planarity instance by routing the boundary of cluster C_i between the levels L^l_i and L^u_i and closing the curve to encompass all vertices on lower levels; see Figure 6.9 (b). Therefore, the order of split edges on a cluster boundary will equal the vertex order on the corresponding levels.

For the converse, we will use the order in which edges cross the boundary of C_i in a clustered-planar drawing as order of the corresponding vertices on levels L^l_i and L^u_i. Contracting cluster C_i into a vertex v_i and $G \setminus C_{i+1}$ into a vertex v_{i+1} (while maintaining the order of edges crossing the respective cluster boundaries) yields a drawing with vertices $L^u_i \cup L^l_{i+1} \cup \{v_i, v_{i+1}\}$; see Figure 6.9 (c). Further contracting L^u_i into v_i and L^l_{i+1} into v_{i+1}, we can obtain a radial level-planar drawing only consisting of parallel edges between v_i and v_{i+1}. We can now iteratively decontract the vertices of L^u_i and L^l_{i+1}, using the same y-coordinate between v_i and v_{i+1} for vertices on the same level. In this way, we obtain a radial level-planar drawing for the edges connecting level L_i to level L_{i+1}. Doing this for every level, the order of the vertices on level L^l_i will match the order of the respective vertices on level L^u_i. Thus, we can combine the individual drawing to obtain a radial level-planar drawing of the whole instance. ■

Strip Planarity [Ang+16; DaL15] is a variant of Level Planarity where levels are not represented by horizontal lines but horizontal strips, which allow their contained vertices to be slightly shifted vertically. The authors of the problem note that it coincides with the Atomic Embeddability problem when its host graph is a path [Ang+16; DaL15]. As Atomic Embeddability is equivalent to Synchronized Planarity, Strip Planarity instances can be directly solved by the algorithm for Synchronized Planarity.

6.4.5 Clustered Level Planarity

In a 2004 paper, Bachmaier and Forster [FB04] study the combination of Clustered and Level Planarity. They pose the additional requirement of cluster boundaries being convex in the resulting drawing, which is why we refer to this variant as Convex Clustered Level Planarity. Bachmaier and Forster showed that Convex Clustered Level Planarity is solvable in linear time if the instance is proper (edges only connect adjacent levels) and level-connected (each cluster contains an edge between any pair of adjacent levels it spans). Angelini et al. [Ang+15a] gave a quadratic algorithm for the non-level-connected but proper case and show that the non-proper case is NP-complete. We are not aware of further work on Convex Clustered Level Planarity or a version of it not requiring cluster convexity. We will give linear-time reduction of Clustered Level Planarity not requiring cluster convexity to Synchronized Planarity when the input is biconnected and has a single source. The resulting instance has the same asymptotic size as the instance stemming only from the clustering, showing that Clustered Level Planarity can be solved in the same time as Clustered Planarity in this case.

▶ **Theorem 6.17.** Single-source biconnected Clustered Level Planarity can be reduced to Synchronized Planarity in linear time. The resulting instance has $O(n)$ pipes of maximum degree $O(n^2)$. ◀

Proof. Brückner and Rutter [BR23b] introduced the so-called LP-tree, a variant of the SPQR-tree that describes all level-planar embeddings of a biconnected graph with a single source. It uses the same types of node and also maintains the property that "embedding choices consist of arbitrarily permuting parallel edges between two poles or choosing the flip of a skeleton whose embedding is unique up to reflection" [BR23b]. The only difference between an SPQR- and an LP-tree is that the rigid nodes of the LP-tree are not necessarily triconnected, but still have a prescribed embedding which may only be mirrored. Thanks to their high similarity, LP-trees can be used as drop-in replacement for SPQR-trees in various algorithms, translating them from a planar to a level planar setting [BR23b]. This also applies

to the algorithm for deriving embedding trees from an SPQR-tree, which can easily be ported to LP-trees: We still translate an occurrence of a vertex in an LP-tree rigid node to a Q-node in its embedding PQ-tree, and an occurrence in a parallel node in the LP-tree to a P-node in the PQ-tree. These embedding trees can now be used to constrain the rotations of their vertices, e.g., in an instance of Partially PQ-constrained Planarity. This instance is not yet equivalent to the input Level Planarity instance, as we additionally need to ensure that all vertices in a non-triconnected rigid node are flipped consistently (as there is no triconnected graph minor that ensures this). Using the Synchronized Planarity reduction of Partially PQ-constrained Planarity this can easily be ensured by putting the Q-nodes stemming from the same LP-tree rigid node into the same partition cell. Note that the underlying graph does ensure that the rotations of P-nodes that represent the poles of the same parallel node of the LP-tree (which represents a part of a parallel node of the SPQR-tree) line up; see [BR16b, Section 4.1] and [BR23b, Section 3.2]. Thus, the restrictions from the leveling can be translated to PQ-trees constraining vertex rotations plus a partition of Q-nodes, where all Q-nodes in the same set need to be flipped consistently.

As the reduction from Clustered- to Synchronized Planarity leaves the vertices of the input graph unchanged, we can annotate the obtained Synchronized Planarity instance with the rotation-constraining embedding trees obtained from the Level Planarity instance. This yields an instance of PQ-constrained Synchronized Planarity where each solution simultaneously corresponds to a solution of both of the input Clustered and Level Planarity instances. Reducing this back to Synchronized Planarity using Theorem 6.20 adds one pipe to each vertex in the input graph. As the reduction of clusters also adds $O(n)$ pipes of maximum degree $O(n^2)$, the resulting instance is not asymptotically larger. The LP-tree required for the reduction can be found in linear time [BR23b]. ■

6.4.6 Connected SEFE-2

For two graphs $G^{①}$ and $G^{②}$, SEFE-2 is equivalent to finding a pair of planar embeddings that induce the same (i.e. consistent) cyclic edge orders and the same (i.e. consistent) relative positions on their shared graph $G = G^{①} \cap G^{②}$ [BKR17; JS09]. Our algorithm can be used to provide the synchronization for the first half of this requirement, which is sufficient for instances with a connected shared graph [BKR13].

▶ **Theorem 6.18.** Connected SEFE-2 can be solved in $O(n^2)$ time. ◀

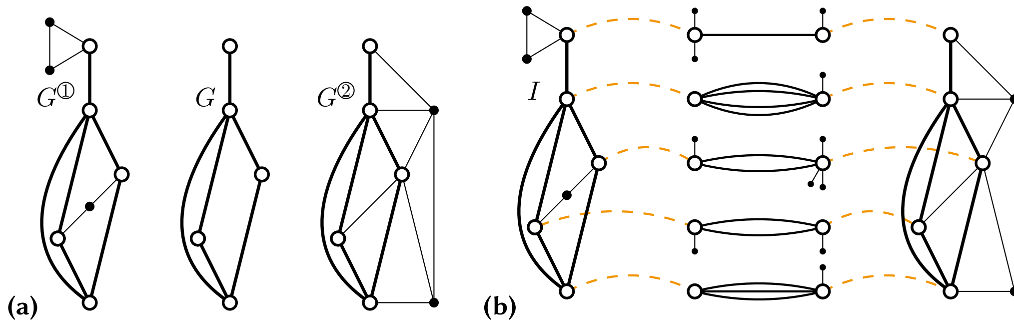


Figure 6.10: An instance of Connected SEFE-2 $G = G^{①} \cap G^{②}$ **(a)** and the equivalent Synchronized Planarity instance I **(b)**.

Proof. We add both $G^{①}$ and $G^{②}$ to the Synchronized Planarity instance and also add a bond with poles $b^{①}$ and $b^{②}$ for each vertex $x \in G$. The parallel edges of the bond correspond to the edges incident to x in G. We add further degree-1 vertices so that we can match $x^{①}$ (i.e. vertex x in $G^{①}$) with $b^{①}$ and $x^{②}$ with $b^{②}$; see Figure 6.10. The Synchronized Planarity algorithm can then be used to obtain embeddings $\mathcal{E}_1$ and $\mathcal{E}_2$ for $G^{①}$ and $G^{②}$, respectively. As pipes reverse the order of incident edges, a solution for the Synchronized Planarity instance will have one of the two graphs mirrored with respect to the graph shared with the other, that is, the solution for the SEFE-2 instance is $\mathcal{E}_1, \overline{\mathcal{E}_2}$. ■

6.4.7 Partially (F)PQ-constrained Planarity

Another problem that investigates the enforcement of rotation constraints in planar embeddings is Partially PQ-constrained Planarity [BR16b]. Here, each vertex in the graph can be annotated with a PQ-tree that limits the rotations of (some of) its incident edges.

▶ **Theorem 6.19.** Partially PQ-constrained Planarity can be solved in $O(m^2)$ time. ◀

Proof. Instances of Partially PQ-constrained Planarity can be converted to equivalent instances of Synchronized Planarity by, for each vertex v, adding its PQ-tree T_v to the graph, converting Q-nodes to Q-vertices similarly to PropagatePQ. Afterwards, for each inserted tree T_v, we add a cap-vertex c_v and connect all leaves of T_v to c_v. To be able to match the cap-vertices with the vertices in the original graph, further degree-1 vertices are connected to the cap-vertices until

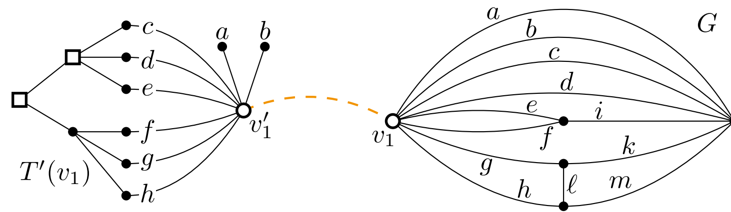


Figure 6.11: The equivalent Synchronized Planarity instance for an instance of Partially PQ-constrained Planarity, where the PQ-tree $T'(v_1)$ (left) restricts the order of the edges $\{a, \ldots, h\}$ around the vertex v_1 in G (right). The cap-vertex v'_1 was added together with two degree-1 vertices.

their degree matches the respective node in the original graph; see Figure 6.11. Adding a pipe matching c_v with v then ensures that the rotation of any vertex v is compatible with the PQ-tree T_v it was annotated with. ■

Note that previous algorithms are geared towards biconnected instances and cannot handle cut-vertices of degree more than 5, while this approach works for general graphs. While the linear-time algorithm for Partially PQ-constrained Planarity by Bläsius and Rutter is restricted to biconnected graphs [BR16b], this allows us to solve general instances. Bläsius and Rutter also note that their algorithm can easily be extended to the case of FPQ-trees, which also contain F-nodes having an entirely fixed rotation without allowing flips. Note that this makes the problem equivalent to EC-Planarity with Free Edges as presented by Gutwenger et al. [GKM08]. We can also handle F-nodes in our reduction by placing all of them in the same partition cell as Q-vertices. If the vertices in this cell now have the wrong flip in a solution, we can simply mirror the embedding and thus ensure that all F-nodes have their single correct rotation.

6.4.8 (F)PQ-Constrained Synchronized Planarity

We now present a variant of Synchronized Planarity that also allows constraints as given in PQ-constrained Planarity for pipes. That is, we are given an instance I of Synchronized Planarity and, for each pipe $\rho = (v, u, \varphi_{vu})$, a PQ-tree $T(\rho)$ where the leaves $L(T(\rho))$ are the edges $E(v)$ incident to its one endpoint v. We seek an embedding $\mathcal{E}$ as solution for I where additionally, for each pipe ρ, the order $\mathcal{E}(v)$ of edges incident to its respective endpoint is admissible by its PQ-tree $T(\rho)$. We show that this problem can be solved by reducing it back to Synchronized Planarity, expressing the additional constraints as we did for

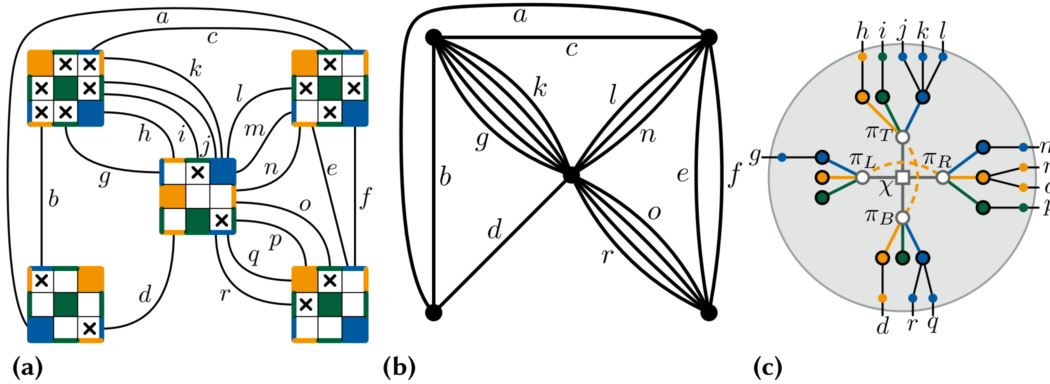


Figure 6.12: An instance of Row-Column Independent NodeTrix Planarity **(a)** and its equipped frame graph, where all clusters were contracted to single vertices **(b)**. The Synchronized Planarity gadget used for representing the middle adjacency matrix **(c)**.

PQ-constrained Planarity. Note that the variant of this problem where the PQ-trees only constrain some of the incident edges is shown to be NP-complete in Section 6.5.3.

▶ **Theorem 6.20.** PQ-constrained Synchronized Planarity can be reduced to Synchronized Planarity in linear time using $O(m)$ pipes of total degree $O(m)$. ◀

Proof. For each pipe $\rho = (v, u, \varphi_{vu})$, we insert two copies of its PQ-tree $T(\rho)$ into the graph, synchronizing their inner nodes using pipes and partition cells as we did during the insertion of PropagatePQ. Similar to the previous section, we connect all leaves of the first copy of the PQ-tree to a new cap-vertex v' and the leaves of the other copy to new cap-vertex u'. Finally, we replace ρ by pipes $\rho_v = (v, v', \varphi_{v,v'})$ and $\rho_u = (u, u', \varphi_{u,u'})$ where the bijections $\varphi_{v,v'}$ and $\varphi_{u,u'}$ are given naturally by the correspondence of leaves to incident edges. Note that this still ensures that the rotations of u and v are synchronized, but now they also need to respect the orders prescribed by the inserted PQ-tree. For each vertex v, we created $O(\deg(v))$ new pipes with a total degree in $O(\deg(v))$. Furthermore, the replacement can be done in $O(\deg(v))$ time. Thus, we obtain an equivalent instance in linear time using $O(m)$ pipes of total degree $O(m)$. ■

This reduction can easily be adapted to also handle FPQ-trees using the same approach as in the previous Section 6.4.7.

6.4.9 NodeTrix Planarity

For the problem NodeTrix Planarity, we are given a graph together with a flat clustering (i.e., a partition) of its vertices and seek a planar representation of the graph where each cluster is replaced by an adjacency matrix of its contained vertices [HFM07]. Thus, edges within the same cluster are represented by an entry in the respective adjacency matrix. Edges between different clusters shall be drawn planarly such that they connect the borders of the adjacency matrices of their endpoints, ending exactly at the border of a row or column representing the respective endpoint. We only consider the "fixed sides scenario", where the input specifies for each edge endpoint whether it should lie on the top, bottom, left or right of the matrix. This problem is NP-complete in the general case, but linear-time solvable if the order of vertices (i.e. rows and columns) for each adjacency matrix is fixed [DaL+18]. We consider the relaxation Row-Column Independent NodeTrix Planarity, where the order of the rows need not match the order of the columns. Liotta et al. [LRT21] show that this problem can be solved in quadratic time if the graph obtained by contracting each cluster into a single vertex is biconnected. Using Synchronized Planarity, the problem can also be solved without this restriction.

▶ **Theorem 6.21.** Row-Column Independent NodeTrix Planarity can be solved in $O(n^2)$ time. ◀

Proof. We construct a Synchronized Planarity gadget for representing each adjacency matrix similarly to Liotta et al. [LRT21], as shown in Figure 6.12. The only difference is that we can use pipes to directly synchronize the orders of the rows (i.e. the rotation of π_L and π_R) and columns (i.e. the rotation of π_T and π_B) on both sides. The gadget center vertices χ are Q-vertices in the same partition cell, so that after solving the Synchronized Planarity instance, we can mirror the embedding if necessary so that π_R is consistently right of π_L. ■

The solution for biconnected instances by Liotta et al. [LRT21] works via a reduction to the more general problem 1-Fixed Constrained Planarity. Later in the following section, we will show that the 1-Fixed Constrained Planarity instances considered by Liotta et al. can also be reduced to Synchronized Planarity, albeit using a more involved reduction than in this section.

6.4.10 Synchronized Planarity with Twisted Pipes and 1-Fixed Constrained Planarity

Finally, we present an extension of Synchronized Planarity that also allows some pipes to be designated as *twisted pipes* where the endpoints no longer have to have opposing rotations under their bijection, but the exact same rotation. While this kind of pipe may sound more natural, it is actually far less natural in terms of the splits and joins from Section 6.3.1 that motivate the correctness of our algorithm. This is because the two vertices obtained from splitting a graph at a cut need to have opposing, not the same rotations, for the two graphs to be joined again; see Figure 6.1.

▶ **Theorem 6.22.** Synchronized Planarity with Twisted Pipes can be solved in $O(m^2)$ time. ◀

Proof. We will show how the individual operations need to be adapted to also apply to twisted pipes. The correctness and running time will follow analogously to the base algorithm without twisted pipes. First, consider the operation `SimplifyMatching`. In the 'terminal' case (i), the pipe is simply removed as one of its endpoint admits any rotation. The same can be done in case of a twisted pipe, simply using the reversed rotation for the unconstrained endpoint when constructing an embedding. In the 'toroidal' case (ii), we check whether the permutation induced by the pipe before its removal is order preserving. The order has to be preserved as traversing the pipe leads to a first order reversal and subsequently traversing the edges of the parallel reverses the order once again. If the pipe is twisted, the order reversals no longer cancel each other out and we need to check for an *order reversing* permutation π with $\pi(O) = \overline{O}$ for any order O instead. Bläsius and Rutter [BR16b, Lemma 2.3] show that a permutation is order reversing if and only if all its permutation cycles have length 2, except for at most two cycles with length 1. If this holds for the permutation induced by the twisted pipe, it can simply be removed; otherwise we return a trivial no-instance. In the transitive case (iii), two pipes matching the poles of a parallel with other vertices are replaced by a single direct pipe. If exactly one of the removed pipes is twisted, so is the replacement pipe. If both removed pipes are twisted, the replacement pipe is not twisted. Again, the order of the edges in the parallel is unconstrained except for the pipes that we removed and can thus be chosen to satisfy the twisted pipes.

If operation `PropagatePQ` is applied on a twisted pipe, the resulting pipes synchronizing the inner P-nodes of the inserted PQ-trees are also twisted. The default rotations of inner Q-nodes are no longer flipped on one side, making them the same in both copies. This ensures that contracting the two trees individually

in a solution yields two vertices with the exact same and not reversed rotations. Finally, for operation `EncapsulateAndJoin`, we will resolve the twistedness of the pipe connecting two cut-vertices after their encapsulation, allowing their join to proceed as usual. Observe that we can mirror the embedding of a single connected component in a solution if the component contains no Q-vertices and we switch all its pipes from being non-twisted to twisted and vice-versa. We mirror an arbitrary one of the two stars generated by the encapsulation, marking all pipes from its rays to blocks as twisted and marking its pipe to the other cut-vertex non-twisted. This yields an equivalent instance in which the two cut-vertices matched by a usual pipe can be joined as usual. ■

We can now use this notion of twisted pipes to also solve restricted cases of the problem 1-Fixed Constrained Planarity [LRT21] introduced by Liotta et al., which is related to the Simultaneous FPQ-Ordering problem introduced by Bläsius and Rutter [BR16b] and generalizes Partially PQ-constrained Planarity. The only previous solution to this problem can only handle biconnected graphs and runs in quadratic time, while our solution can handle arbitrary graphs, albeit only with certain constraints. In 1-Fixed Constrained Planarity we are given a planar graph $G = (V, E)$ together with, for every vertex $v \in V$, a so-called 1-fixed constraint $C(v)$, that is a 1-fixed instance of Simultaneous FPQ-Ordering. An instance of Simultaneous FPQ-Ordering [BR16b] consists of a DAG $D = (N, A)$ with nodes $N = \{T_1, \dots, T_k\}$, each one being a FPQ-tree. Each arc from a node T_j to a node T_i has an injective mapping $\phi : L(T_j) \to L(T_i)$. A solution to Simultaneous FPQ-Ordering consists of cyclic orders $\sigma_1, \dots, \sigma_k$ such that σ_i is admissible by T_i for $i \in \{1, \dots, k\}$ and for each arc (T_i, T_j, ϕ), $\phi(\sigma_j)$ is a suborder of σ_i (which is conversely also denoted as σ_i *extending* $\phi(\sigma_j)$). Bläsius and Rutter [BR16b] slightly extend this by also allowing so-called *reversing arcs*, where it instead has to hold that $\phi(\sigma_j)$ is a suborder of $\overline{\sigma_i}$.

In 1-Fixed Constrained Planarity, the 1-fixed constraint $C(v)$ for any vertex v of graph G has a single source node, which is a FPQ-tree whose leaves correspond to the edges incident to v. A solution to the problem consists of a planar embedding of G together with a solution for each constraint $C(v)$, where the cyclic order σ_1 of the single source node T_1 of $C(v)$ is the rotation of v. The constraints in 1-Fixed Constrained Planarity need to satisfy two properties: First, every node whose FPQ-tree contains a P-node has at most 2 parents. Second, the interactions between P-nodes of different FPQ-trees connected in the DAG has to be limited in a sense defined by the concept of fixedness. For two inner nodes μ_i, μ_j of FPQ-trees T_i, T_j, respectively, we say that μ_i *is fixed by* μ_j (and conversely that μ_j *fixes* μ_i) if there are leaves $x, y, z \in L(T_j)$ such that (i) deleting μ_j from T_j pairwise disconnects $x, y,$

and z, and (ii) deleting μ_i from T_i pairwise disconnects $\phi(x)$, $\phi(y)$, and $\phi(z)$. Bläsius and Rutter [BR16b] show that for any arc (T_i, T_j, ϕ) and any inner node node μ_j of T_j, we can assume the FPQ-tree T_i to contain exactly one node μ_i that is fixed by μ_j, making the instance *normalized*.

In our 1-fixed instance of Simultaneous FPQ-Ordering, every P-node μ_i of a FPQ-tree T_i is conversely fixed by at most one P-node μ_j of a child T_j of T_i.[9] This means that every P-node fixes at most two P-nodes in its (up to two) parents while being fixed by at most one P-node in a child. Unfortunately, this synchronization of one P-node with up to three different other P-nodes is hard to model using pipes. For our reduction to Synchronized Planarity, we will add a further restriction requiring a P-node that is fixed by a P-node of a child to fix at most one P-node in a parent. We call this variant *strict* 1-Fixed Constrained Planarity. Thanks to these restrictions, it is now easy to reduce the problem to Synchronized Planarity, as every P-node needs synchronization with at most two other nodes. We note that all applications described by Liotta et al. [LRT21] yield instances that are strict.

▶ **Theorem 6.23.** Strict 1-Fixed Constrained Planarity can be reduced to Synchronized Planarity with Twisted Pipes in linear time. ◀

Proof. We copy the graph G underlying the 1-Fixed Constrained Planarity instance into the Synchronized Planarity instance $\mathcal{I}$. We will translate each constraint $C(v)$ constraining a vertex v in G into further components modeling the constraint, which we can then synchronize with v via a pipe. For each FPQ-tree T_i in $C(v)$ we add two copies T_i', T_i'' to $\mathcal{I}$, turning P- and Q-nodes into P- and Q-vertices, respectively, and keeping the default rotations of the latter. For each leaf ℓ of T_i, we identify the respective leaves ℓ', ℓ'' of T_i', T_i'' with each other. For each tree, this yields one connected component which contains two copies of each inner node. In a planar embedding, these two copies will have a reversed rotation due to the graph structure that connects them. The only exception to this is the single source T_1 of $C(v)$, for which we only add one copy T_1'' and connect all its leaves to a single new node v'. Again, the rotation of v' will correspond to an admissible order of T_1 in any planar embedding.

It remains to model the arcs of $C(v)$ through pipes and Q-constraints. Recall that because we can assume $C(v)$ to be normalized, each inner node of a FPQ-tree fixes exactly one inner node of each of its (at most two) parents. Due to the 1-

9 Note that we here use the older definition of (1-)fixedness on normalized instances as given in [BR16b], which is not as generic as the newer definition not requiring normalization from [LRT21]. While the later one is easier to compute for arbitrary instances, the former one is more restrictive, making it easier to work with when solving instances. Any instance can be made normalized while maintaining its fixedness, making both definitions equivalent.

fixedness, each inner P-node of a FPQ-tree is also fixed by at most one P-node of one of its children. For each node μ_i of tree T_i, we have a copy μ' that can be used for synchronization with a node of one parent of T_i and a copy μ'' (with reversed rotation) that can either be used for synchronization with a node of the one child or of the other parent of T_i. We will first handle the case of nodes with at most one parent. To handle Q-nodes fixing Q-nodes, we simply add both to the same partition cell to synchronize their rotations. To handle Q-nodes fixing P-nodes (or vice-versa), we replace the Q-node with a wheel and synchronize its center with the P-node. We turn one of the vertices on the rim of the wheel into a Q-node synchronizing the flip of the wheel with the partition cell of the former Q-node. To handle P-nodes fixing P-nodes, we match both by a pipe. For nodes with two parents but no children, we will handle the synchronization with the first parent as before using μ'. For the second parent, we will also use the same approach, but now synchronize with μ'', which has no other pipe as μ_i is not fixed by another P-node of a child of T_i in our strict instance. Finally, we add a pipe between v and v'. Traversing this pipe and then the edges from v' to T_1'' will reverse the order twice, making the rotation of v the same as the order of leaves of T_1'' in any solution to the Synchronized Planarity instance. We mark all remaining pipes as twisted depending on whether the corresponding arc is reversing or not. ■

6.5 Related NP-hard Problems

Having seen various applications of Synchronized Planarity for efficiently solving problems that lie "above" in the hierarchy of Figure 3.1, we will now have a look at further problems "below" Synchronized Planarity in this hierarchy. First, we will position our problem directly above many of the problems in the NP-complete section of the hierarchy by reducing it to SEFE-2. Afterwards, we will show that Synchronized Planarity is in a sense close to being NP-complete by showing that further increasing its modeling capacities by allowing multiple pipes per vertex or by adding partial pipe constraints makes the problem NP-complete.

6.5.1 Reduction to SEFE-2

To complement the reductions from the previous section giving polynomial-time solutions for other problems, we give a reduction of Synchronized Planarity to the problem SEFE-2, whose complexity is yet unknown. In a sense, this makes

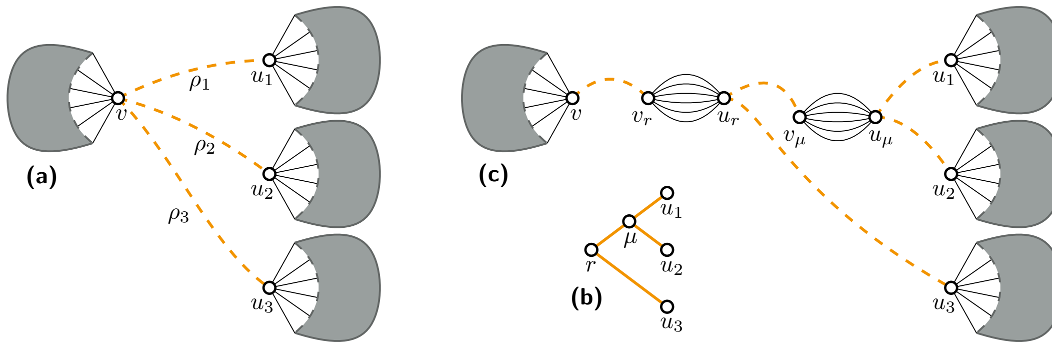


Figure 6.13: **(a)** A P-vertex v that is endpoint of $k = 3$ pipes ρ_1, ρ_2, ρ_3 with other endpoints u_1, u_2, u_3. **(b)** The binary tree T with leaves u_1, u_2, u_3, root r and an inner node μ. **(c)** The replacement for the pipes ρ_1, ρ_2, ρ_3.

Synchronized Planarity the direct "interface" between the SEFE-2 problem in the "unknown complexity" region of our hierarchy and most of the problems in the "efficiently solvable" (P) region in Figure 3.1.

▶ **Theorem 6.24.** Synchronized Planarity can be reduced to SEFE-2 in linear time. ◀

Proof. We subdivide each edge twice, moving the resulting middle segments to one exclusive graph and the begin and end segments to the shared graph, which thus, for each vertex, contains a star of the same degree. The other exclusive graph then encodes the constraints of all pipes and partition cells as follows. For each pipe, the shared edges incident to its endpoints are connected according to the bijection of the pipe. The edges incident to all Q-vertices in the same partition cell are connected to form a triconnected component, respecting the rotations enforced by the Q-vertices. The bonds in the second exclusive graph now ensure that all pipes are respected, while the rigids ensure that the rotations of Q-vertices are respected and in sync. Note that the shared graph only contains stars in the resulting SEFE-2 instance, thus we do not need to consider relative positions. ■

6.5.2 Multi-Synchronized Planarity

In the generalization k-Synchronized Planarity, each P-vertex can be endpoint of at most k pipes (each in turn still having exactly 2 endpoints). Note that 1-Synchronized Planarity is the equivalent to the problem as we studied it up to now,

while solving 0-Synchronized Planarity is equivalent to solving a pipe-free, i.e., reduced instance using Lemma 6.8. We first show that allowing values of k greater than 2 does not increase the complexity of the problem.

▶ **Lemma 6.25.** *k-Synchronized Planarity can be reduced to 2-Synchronized Planarity in polynomial time.* ◀

Proof. Let v be a P-vertex that is endpoint of the $k \geq 3$ pipes $\rho_1, \ldots, \rho_k$ with other endpoints $u_1, \ldots, u_k$, respectively; see Figure 6.13. Let T be a rooted binary tree with leaves $u_1, \ldots, u_k$ where all inner nodes (including the root) have degree at most 3. For each inner node μ of T, we add a bond with poles v_μ, u_μ connected by $\deg(v)$ edges, each corresponding to one edge incident to v. Additionally, for each node μ with parent ν, we add one pipe matching v_μ with u_ν, deriving the bijections from the correspondence of edges of the bond to edges incident to v. For the root node r of T, we add a pipe matching v_r with v. For each leaf u_i with parent ν in T, we add a pipe matching u_ν with u_i. Finally, remove the pipes $\rho_1, \ldots, \rho_k$. Note that this leaves v with only with one pipe while only introducing new vertices and pipes such that one vertex is part of at most two pipes; see Figure 6.13. Furthermore, we only replaced one pipe of each u_i with a single other one. Thus, exhaustively applying this process yields an equivalent 2-Synchronized Planarity instance. ■

Now we will show that 2-Synchronized Planarity (and thus also k-Synchronized Planarity for $k > 1$) is NP-complete, even if either the total number of pipes or the maximum degree of pipes is very limited.

▶ **Theorem 6.26.** *2-Synchronized Planarity is NP-complete, even if the instance contains only two pipes.* ◀

Proof. Obviously, k-Synchronized Planarity can be solved using a guess and verify approach, guessing an embedding and verifying in polynomial time its planarity and that all constraints are satisfied. Angelini et al. [ADN15] show that the NP-hard problem Betweenness [Opa79] can be reduced to SEFE-3 with a star as shared graph; see also the summary in [Rut20]. We will reduce this class of SEFE-3 instances to instances of 2-Synchronized Planarity, showing the NP-hardness of the problem. Note that to find a solution for such a SEFE-3 instance, as the star used as shared graph is connected, it suffices to find planar embeddings of the three exclusive graphs which have consistent rotations for the center of the star [BKR17; JS09]. To do so, we will add the three exclusive graphs separately to an instance of 2-Synchronized Planarity and synchronize their three centers using two pipes. A solution for the resulting 2-Synchronized Planarity instance now corresponds to a solution to the SEFE-3 instance, which corresponds to a solution for the input Betweenness instance. ■

Using a recent result from joint work with Rutter and Pfretzschner [5], we can also show that the case of restricted pipe degrees is NP-hard.

▶ **Corollary 6.27.** 2-Synchronized Planarity is NP-complete, even if the instance contains only pipes of degree 4. ◀

Proof. It can be shown that SEFE with a variable k is NP-complete, even if the shared graph is connected and has maximum degree 4 [5]. Similar to Theorem 6.26 it suffices to ensure consistent rotations for such instances [BKR17; JS09], which can easily be done using separate exclusive graphs with vertices synchronized by pipes. Using Lemma 6.25, we can turn the resulting k-Synchronized Planarity instance into an instance of 2-Synchronized Planarity. ■

6.5.3 Partially FPQ-Constrained Synchronized Planarity

In Section 6.4.7, we showed that partial PQ-constraints to vertices as in Partially PQ-constrained Planarity can be expressed in terms of Synchronized Planarity. In Section 6.4.8, we showed that the same can be done with PQ-constraints to pipes, as long as all incident edges are part of the constraint. We now show that in contrast, partial PQ-constraints to pipes turn the problem NP-complete. The problem Partially PQ-constrained Synchronized Planarity generalizes PQ-constrained Synchronized Planarity and also allows the PQ-trees constraining rotations of pipe endpoints to only constrain some, but not all of the incident edges.

▶ **Theorem 6.28.** Partially PQ-constrained Synchronized Planarity is NP-complete. ◀

Proof. Again, the problem can easily be solved using a guess and verify approach, showing containment in NP. To show NP-hardness, we use a reduction from the NP-hard problem Betweenness [Opa79]. In an instance (X, C) of this problem, we seek a linear order σ of a base set X such that for each triple $(x, y, z) \in C \subset X^3$, y lies between x and z in σ. Note that the ordering of x and z is arbitrary and the elements x, y, and z need not be consecutive. As Synchronized Planarity works with cyclic orders instead of linear orders, we will add a separator element ℓ to X and work with the resulting set X'. We start building a corresponding instance of Partially PQ-constrained Synchronized Planarity by adding a bundle of $|X'|$ edges, each one corresponding to a different element of X'. From this bundle we will build a chain by, for each constraint $(x, y, z) \in C$, adding a further bundle of $|X'|$ edges and synchronizing one of the endpoints of the chain with one of the endpoints of the newly inserted bundle. This pipe has a constraint consisting of a single Q-node with leaves $[\ell, x, y, z]$, effectively allowing the positions of x and z to

be swapped with regard to y but ensuring that y is between x and z when cutting the cyclic order at ℓ. Note that due to the pipes, in a solution to the generated instance, all bundles have the same order of edges and the fixed rotations of all Q-nodes are respected. Thus, we can obtain a linear order that is a solution to (X, C) from any solution to our generated instance by cutting the cyclic order that is the rotation of any one of the vertices in the built chain at ℓ. Conversely, it is easy to see that appending ℓ and interpreting a linear order that is a solution to Betweenness as cyclic yields valid rotations for the Partially PQ-constrained Synchronized Planarity instance. ∎

6.6 Comparison with the Fulek-Tóth Algorithm

We want to point out that there are many parallels between our algorithm for solving Synchronized Planarity and Fulek and Tóth's solution to Atomic Embeddability. First note that both problems are linear-time equivalent, a reduction in the one direction has been given in the previous section. To reduce from Synchronized Planarity to Atomic Embeddability, all connected components can be turned into atoms, subdividing pipes that loop back to the same component with a trivial atom. It remains to encode the Q-constraints, which can be done by converting all Q-vertices to wheels and synchronizing the centers for all Q-vertices in the same partition cell with pipes to one additional atom per cell. Using a triconnected graph for this atom ensures that exactly two flips are possible and all Q-vertices in the cell are flipped the same way.

The algorithm of Fulek and Tóth relies on seven basic operations:

- `Suppress` eliminates pipes with degree two or less,
- `Delete` removes certain edges when all affected components are subcubic,
- `Split` splits an atom into its connected components, and
- `Detach` splits unmatched cut-vertices.
- `Enclose` encloses a cut-vertex inside its own atom,
- `Contract` joins two neighboring atoms (under certain conditions), and
- `Stretch` encodes the fact that a subset of edges incident to a vertex must be consecutive by splitting the vertex into two.

Fulek and Tóth carefully orchestrate these operations into two large subroutines, which they then iteratively use to globally reduce the maximum degree of cut-vertices that correspond to pipes. Eventually all such cut-vertices have small degree and the problem can be solved directly as each part of the instance is either subcubic or what they call toroidal.

Our algorithm only uses four basic operations: ConvertSmall, EncapsulateAndJoin, SimplifyMatching, and PropagatePQ. These operations can be applied in an arbitrary order to reach a reduced instance, which can then be solved directly.

In direct comparison, our operation ConvertSmall handles all the cleanup (and more) that Fulek and Tóth achieve with Suppress and Delete. Further, in our context, the operation Split is not needed, since we do not differentiate between connected components and atoms, and Detach is not needed since we handle unmatched cut-vertices in our base case algorithm rather than during the reduction phase.

Our operation EncapsulateAndJoin replaces multiple calls to Enclose and Contract as a much more targeted solution. The former also encompasses the stretching of local branches in Carmesin's work [Car17c]. Our operation and the conditions under which it can (and must be) applied clearly expose the key insight that also stands behind the algorithm of Fulek and Tóth, and which we have formalized in Lemma 6.2: Any planar embedding of the bipartite graph resulting from contracting two stars at their centers respects the cut made for splitting both stars. Furthermore, there are parallels between Contract and case (iii) of SimplifyMatching (i.e. when a bond links two distinct pipes).[10] Finally, thanks to the use of PQ- and SPQR-trees, PropagatePQ can be seen as a much more focused replacement of multiple iterations of Stretch. An example for how the operations relate can be seen in Figure 11 of the paper by Fulek and Tóth [FT22], where an instance on which SimplifyMatching and then PropagatePQ can (and clearly should) be applied, is first Contracted twice in Step (iv.a) and then Stretched in (a possibly later iteration of) Step (v.c) of their Subroutine 2.

To conclude, our approach makes it clear to see where progress is made, uncovering important ideas that are not obvious in the global degree-reduction approach employed by Fulek and Tóth.

6.7 Conclusion

In this chapter, we give a quadratic-time algorithm for Synchronized Planarity, which improves the previous $O(m^8)$-time algorithm for the linear-time equivalent problem Atomic Embeddability [FT22]. Similar to Goldberg and Tarjan's

10 Contract could also be used instead of PropagatePQ between two block-vertices of distinct connected components, although this would break our running time analysis; see Section 9.3.2.

push-relabel algorithm, it relies on few and simple operations that can be applied in an arbitrary order. They also highlight where and how progress is made and thereby clearly expose key ideas that also underlie the algorithm for Atomic Embeddability.

The applications of Synchronized Planarity include solving Clustered Planarity, Connected SEFE-2, Partially PQ-constrained Planarity, Row-Column Independent NodeTrix Planarity and various other constrained planarity variants in quadratic time, thanks to linear-time reductions to Synchronized Planarity for all of them. For Clustered Planarity, this improves over the previously fastest algorithms via the $O(m^8)$-time algorithm for Atomic Embeddability. In the case of Connected SEFE-2 the reduction used by Fulek and Tóth [FT22] includes a quadratic blowup and therefore yields an $O(n^{16})$-time algorithm. Our direct linear-time reduction leads to a quadratic algorithm. Partially PQ-constrained Planarity and Row-Column Independent NodeTrix Planarity were only solved for biconnected instances. Our algorithms achieve the same quadratic running time, but work for all instances. Altogether, we show that almost all constrained planarity variants for which efficient solutions are known can be solved via a reduction to Synchronized Planarity; see Figure 3.1. Many of these reductions are very natural, showing that Synchronized Planarity serves as a useful tool when modelling or solving constrained planarity problems.

While our algorithm efficiently solves Connected SEFE-2, the complexity of the general SEFE-2 problem, where the shared graph is not necessarily connected, remains unknown. Jünger and Schulz showed that SEFE-2 is equivalent to finding a pair of planar embeddings that induce the same (i.e. consistent) cyclic edge orders and the same (i.e. consistent) relative positions on the common graph [BKR17; JS09]. Our algorithm can be used to provide the synchronization for the first half of this requirement, which is sufficient for instances with a connected shared graph. It would be interesting to investigate whether Synchronized Planarity can be extended, e.g. using cycle bases of the shared graph [BKR17; BR15], to also ensure consistent relative positions in the common graph and thus solve instances where the shared graph is not connected.

7 Maintaining Triconnected Components under Node Expansion

This chapter is based on joint work with Ignaz Rutter, which also appeared at EuroCG 2023 [9] and CIAC 2023 [6].

The SPQR-tree is a data structure that represents the decomposition of a graph at its *separation pairs*, that is the pairs of vertices whose removal disconnects the graph. The components obtained by this decomposition are called *skeletons*. SPQR-trees form a central component of many graph visualization techniques and are used for, e.g., planarity testing and variations thereof [DT96b; HR20a] and for computing embeddings and layouts [Gut10; Wei02]; see [Mut03] for a survey of graph drawing applications. Outside of graph visualization they are used in the context of, e.g., minimum spanning trees [BM89; DT96b], triangulations [BKK97], and crossing optimization [Gut10; Wei02]. They also have multiple applications outside of graph theory and even computer science, e.g. for creating integrated circuits [CHH99; Zha+13], business processes modelling [VVK09], electrical engineering [FOO05], theoretical physics [MS12] and genomics [Fed+17].

Initially, SPQR-trees were devised by Di Battista and Tamassia for incremental planarity testing [DT89; DT96b]. As such, even in their initial form, SPQR-trees already allowed dynamic updates in the form of edge addition. Their use was quickly expanded to other on-line problems [DT90; DT96a] and to the fully-dynamic setting, which allows insertion and deletion of vertices and edges in $O(\sqrt{n})$ time [Epp+96], where n is the number of vertices in the graph. The original descriptions of SPQR-trees by Di Battista and Tamassia were first published in 1989 [DT89] and 1990 [DT90] and are based on ideas by Bienstock and Monma [BM89; BM90], while the corresponding full versions were published in 1996 [DT96a; DT96b]. In the meantime, the algorithms for incrementally maintaining triconnectivity and planarity information were already improved by Westbrook [Wes92] and La Poutré [Pou92; Pou94], obtaining an amortized running time of $O(\alpha(q, n))$ for q operations where α is the inverse Ackermann function. Recently, Holm and Rotenberg describe a fully-dynamic algorithm for maintaining planarity information in $O(\log^3 n)$ time per operation [HR20a; HR20b]. See also there for a short overview over more recent usages of SPQR-trees.

In this chapter, we consider an incremental setting where we allow a single operation that expands a vertex v into a connected graph G_v. This expansion can for

example be the join of a Synchronized Planarity pipe as described in Section 6.3.1 or the operation PropagatePQ from Section 6.3.4, which replaces a vertex with a PQ-tree (where Q-nodes are represented by wheels). The approach of Eppstein et al. [Epp+96] allows this in $O((\deg(v)+|G_v|)\cdot\sqrt{n})$ time by only representing parts of triconnected components. We improve this to $O(\deg(v)+|G_v|)$ using an algorithm that is much simpler and explicitly yields full triconnected components, which will become important for our applications later. In addition, our approach also allows to efficiently merge two SPQR-trees as follows. Given two biconnected graphs G_1, G_2 containing vertices v_1, v_2, respectively, together with a bijection between their incident edges, we construct a new graph G by replacing v_1 with $G_2 - v_2$ in G_1, identifying edges using the given bijection. Given the SPQR-trees of G_1 and G_2, we show that the SPQR-tree of G can be found in $O(\deg(v_1))$ time. For example, this allows an even more efficient join of Synchronized Planarity pipes when both endpoints are block-vertices. To summarize, we present a data structure that supports the following operations: $\texttt{InsertGraph}_{\text{SPQR}}$ expands a single vertex in time linear in the size of the expanded subgraph, $\texttt{Merge}_{\text{SPQR}}$ merges two SPQR-trees in time linear in the degree of the replaced vertices, IsPlanar indicates whether the currently represented graph is planar in constant time, and Rotation yields one of the two possible planar rotations of a vertex in a triconnected skeleton in constant time. Furthermore, our data structure can be adapted to yield consistent planar embeddings for all triconnected skeletons and to test for the existence of three disjoint paths between two arbitrary vertices with an additional factor of $\alpha(n)$ for all operations, where α is the inverse Ackermann function.

The main idea of our approach is that the subtree of the SPQR-tree affected by expanding a vertex v has size linear in the degree of v, but may contain arbitrarily large skeletons. In a "non-normalized" version of an SPQR-tree, the affected cycle ('S') skeletons can easily be split to have a constant size, while we develop a custom splitting operation to limit the size of triconnected ('R') skeletons. This limits the size of the affected structure to be linear in the degree of v and allows us to perform the expansion efficiently.

In addition to the description of this data structure, the technical contribution of our work is twofold: First, we develop an axiomatic definition of the decomposition at separation pairs, putting the SPQR-tree as "mechanical" data structure into focus instead of relying on and working along a given graph structure. As a result, we can deduce the represented graph from the data structure instead of computing the data structure from the graph. This allows us to make more or less arbitrary changes to the data structure (respecting its consistency criteria) and observe how the graph changes, instead of having to reason which changes to the graph require which updates to the data structure.

Problem	Running Times		
	before	basic	improved
Atomic Embeddability / Synchronized Planarity	$O(m^8)$ [FT22]	$O(m^2)$	$O(m \cdot \Delta)$
Clustered Planarity	$O((n+d)^8)$ [FT22]	$O((n+d)^2)$	$O(n+d \cdot \Delta)$
Connected SEFE-2	$O(n^{16})$ [FT22] bicon: $O(n^2)$ [BR16b]	$O(n^2)$	$O(n \cdot \Delta)$
Partially PQ-constrained Planarity	bicon: $O(m)$ [BR16b]	$O(m^2)$	$O(m \cdot \Delta)$
Strip Planarity	$O(n^8)$ [Ang+16; FT22] fix-e.: $O(n^2)$ [Ang+16]	$O(n^2)$	$O(n \cdot \Delta)$

Table 7.1: The best known running times for various constrained planarity problems *before* Synchronized Planarity was published; using the *basic* algorithm as described in Chapter 6; and using the *improved* version with the speed-up from this chapter. Running times prefixed with “bicon” only apply for certain problem instances which expose some form of biconnectivity, while the “fix-e” time only applies in the fixed embedding case. The variables n and m refer to the number of vertices and edges of the problem instance, respectively. The variable d refers to the number of edge-cluster boundary crossings in Clustered Planarity instances, while Δ refers to the maximum pipe degree in the corresponding Synchronized Planarity instances. This is bounded by the maximum number of edges crossing a single cluster border or the maximum vertex degree in the input instance, depending on the problem.

Second, we explain how our data structure can be used to improve the running time of the algorithm for solving Synchronized Planarity from Chapter 6 from $O(m^2)$ (see Theorem 6.12) to $O(m \cdot \Delta)$, where Δ is the maximum pipe degree (i.e. the maximum degree of a vertex with synchronization constraints that enforce its rotation to be the same as that of another vertex). This speed-up also pertains to further constrained planarity variants; see Table 7.1 for an overview of the resulting improvements. Among them is the notorious Clustered Planarity, where we improve the running time from $O((n+d)^2)$ (see Theorem 6.14) to $O(n + d \cdot \Delta)$, where d is the total number of crossings between cluster borders and edges and Δ is the maximum number of edge crossings on a single cluster border.

The *expansion* that is central to this work is formally defined as follows. Let G_α, G_β be two graphs where G_α contains a vertex u and G_β contains $|N(u)|$ marked vertices, together with a bijection ϕ between the neighbors of u and the marked vertices in G_β. With $G_\alpha[u \rightarrow_\phi G_\beta]$ we denote the graph that is obtained from the disjoint union of G_α, G_β by identifying each neighbor x of u with its respective marked vertex $\phi(x)$ in G_β and removing u; see Figure 7.4 for an example. We also say that $G_\alpha[u \rightarrow_\phi G_\beta]$ is a modified version of G_α, where the vertex u was *expanded* into G_β. Note that the expansion generalizes the join from Section 6.3.1.

This chapter is structured as follows. We describe the skeleton decomposition and show how it relates to the SPQR-tree in Section 7.1. The following Section 7.2 extends this data structure by the capability of splitting triconnected components. In Section 7.3, we use this to ensure the affected part of the SPQR-tree is small when we replace a vertex with a new graph. Section 7.4 shows how our results can be used to reduce the time required for solving Synchronized Planarity, Clustered Planarity and related constrained planarity variants.

7.1 Skeleton Decompositions

A *skeleton structure* $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE})$ that *represents* a graph $G_\mathcal{S} = (V, E)$ consists of a set $\mathcal{G}$ of disjoint *skeleton* graphs together with three total, surjective mappings twinE, origE, and origV that satisfy the following conditions:

- Each skeleton $G_\mu = (V_\mu, E_\mu^{\text{real}} \cup E_\mu^{\text{virt}})$ in $\mathcal{G}$ is a multi-graph where each edge is either in E_μ^{real} and thus called *real* or in E_μ^{virt} and thus called *virtual*.
- Bijection $\text{twinE} : E^{\text{virt}} \rightarrow E^{\text{virt}}$ matches all virtual edges in $E^{\text{virt}} = \bigcup_\mu E_\mu^{\text{virt}}$ such that $\text{twinE}(e) \neq e$ and $\text{twinE}^2 = \text{id}$.
- Surjection $\text{origV} : \bigcup_\mu V_\mu \rightarrow V$ maps all skeleton vertices to graph vertices.
- Bijection $\text{origE} : \bigcup_\mu E_\mu^{\text{real}} \rightarrow E$ maps all real edges to the graph edge set E.

Note that each vertex and each edge of each skeleton is in the domain of exactly one of the three mappings. As the mappings are surjective, V and E are exactly the images of origV and origE. For each vertex $v \in G_{\mathcal{S}}$, the skeletons that contain an *allocation vertex* v' with $\text{origV}(v') = v$ are called the *allocation skeletons* of v. Furthermore, let $T_{\mathcal{S}}$ be the graph where each node μ corresponds to a skeleton G_μ of $\mathcal{G}$. Two nodes of $T_{\mathcal{S}}$ are adjacent if their skeletons contain a pair of virtual edges matched with each other. We call a skeleton structure a *skeleton decomposition* if it satisfies the following conditions:

1 (bicon) Each skeleton is biconnected.

2 (tree) Graph $T_{\mathcal{S}}$ is simple, loop-free, connected and acyclic, i.e., a tree.

3 (orig-inj) For each skeleton G_μ, the restriction $\text{origV}|_{V_\mu}$ is injective.

4 (orig-real) For each real edge uv, the endpoints of $\text{origE}(uv)$ are $\text{origV}(u)$ and $\text{origV}(v)$.

5 (orig-virt) Let uv and $u'v'$ be two virtual edges with $uv = \text{twinE}(u'v')$. For their respective skeletons G_μ and G'_μ (where μ and μ' are adjacent in $T_{\mathcal{S}}$), it is $\text{origV}(V_\mu) \cap \text{origV}(V_{\mu'}) = \text{origV}(\{u, v\}) = \text{origV}(\{u', v'\})$.

6 (subgraph) The allocation skeletons of any vertex of $G_{\mathcal{S}}$ form a connected subgraph of $T_{\mathcal{S}}$.

Figure 7.1 shows an example of $\mathcal{S}$, $G_{\mathcal{S}}$, and $T_{\mathcal{S}}$. We call a skeleton decomposition with only one skeleton G_μ *trivial*. In this case, G_μ is isomorphic to $G_{\mathcal{S}}$, and origE and origV are actually bijections between the edges and vertices of both graphs.

To model the decomposition into triconnected components, we define the operations SplitSeparationPair and its converse, JoinSeparationPair, on a skeleton decomposition $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE})$; see also Figure 7.2. For SplitSeparationPair, let u, v be a separation pair of skeleton G_μ and let (A, B) be a non-trivial bipartition of the bridges between u and v.[11] Applying SplitSeparationPair$(\mathcal{S}, (u, v), (A, B))$ yields skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$ as follows. In $\mathcal{G}'$, we replace G_μ by two skeletons G_α, G_β, where G_α is obtained from $G_\mu[A]$ by adding a new virtual edge e_α between u and v. The same respectively applies to G_β with $G_\mu[B]$ and e_β. We set $\text{twinE}'(e_\alpha) = e_\beta$ and $\text{twinE}'(e_\beta) = e_\alpha$. Note that origV maps the endpoints of e_α and e_β to the same vertices. All other skeletons and their mappings remain unchanged.

11 Recall that bridges are the maximal subgraphs which cannot be disconnected by removing or splitting the vertices of a corresponding separation pair. Note that a bridge might consist out of a single edge between u and v and that each bridge includes the vertices u and v.

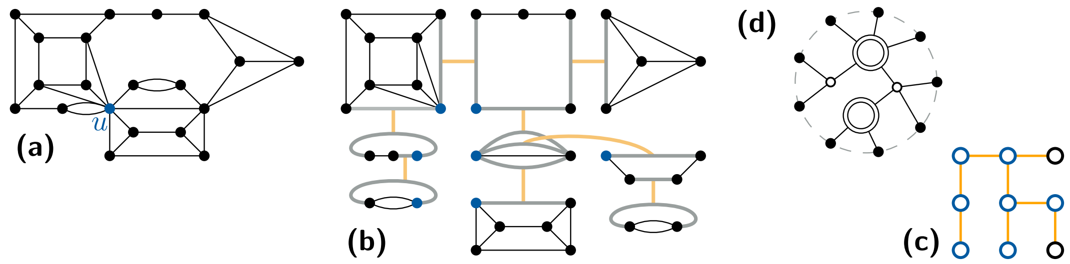


Figure 7.1: Different views on the skeleton decomposition $\mathcal{S}$. **(a)** The graph $G_{\mathcal{S}}$ with a vertex u marked in blue. **(b)** The skeletons of $\mathcal{G}$. Virtual edges are drawn in gray with their matching twinE being shown in orange. The allocation vertices of u are marked in blue. **(c)** The tree $T_{\mathcal{S}}$. The allocation skeletons of u are marked in blue. **(d)** The embedding tree of vertex u as described in Section 7.4.1. P-nodes are shown as white disks, C-nodes are shown as large double disks. The leaves of the embedding tree correspond to the edges incident to u.

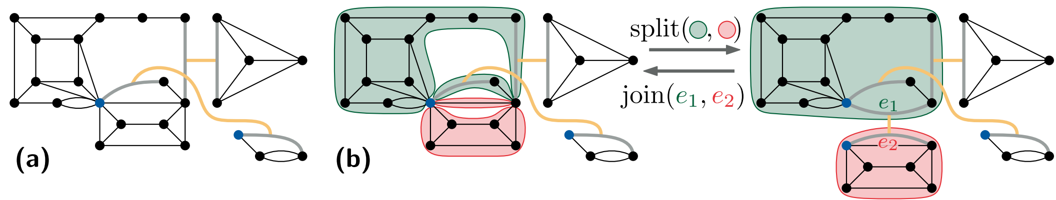


Figure 7.2: (a) A skeleton decomposition that represents graph $G_{\mathcal{S}}$ from Figure 7.1 (a) with the two allocation vertices of graph vertex u marked in blue. **(b)** The result of applying Split-SeparationPair to separate the bridges highlighted in green from those in red. Applying the converse operation JoinSeparationPair on the resulting edges yields the original skeleton.

For JoinSeparationPair, consider virtual edges e_α, e_β with $\text{twinE}(e_\alpha) = e_\beta$ and let $G_\beta \neq G_\alpha$ be their respective skeletons. Applying JoinSeparationPair$(\mathcal{S}, e_\alpha)$ yields a skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$ as follows. In $\mathcal{G}'$, we merge G_α with G_β to form a new skeleton G_μ by identifying the endpoints of e_α and e_β that map to the same vertex of $G_{\mathcal{S}}$. Additionally, we remove e_α and e_β. All other skeletons and their mappings remain unchanged.

The main feature of both operations is that they leave the graph represented by the skeleton decomposition unaffected while splitting a node or contracting an edge in $T_{\mathcal{S}}$, which can be verified by checking the individual conditions.

▶ **Lemma 7.1.** Applying SplitSeparationPair or JoinSeparationPair on a skeleton decomposition $\mathcal{S}$ yields a skeleton decomposition $\mathcal{S}'$ with an unchanged represented graph $G_{\mathcal{S}'} = G_{\mathcal{S}}$. ◀

Proof. We first check that all conditions still hold in the skeleton decomposition $\mathcal{S}'$ returned by SplitSeparationPair. As (A, B) is a non-trivial bipartition, each set contains at least one bridge. Together with e_α (and e_β), this bridge ensures that G_α (and G_β) remain biconnected, satisfying condition 1 (bicon). The operation splits a node μ of $T_\mathcal{S}$ into two adjacent nodes α, β, whose neighbors are defined exactly by the virtual edges in A, B, respectively. Thus, condition 2 (tree) remains satisfied. The mappings origV′, origE′ and twinE′ obviously still satisfy conditions 3 (orig-inj) and 4 (orig-real). We duplicated exactly two nodes, u and v of adjacent skeletons G_α and G_β. Because 3 (orig-inj) holds for G_μ, G_α and G_β share no other vertices that map to the same vertex of $G_{\mathcal{S}'}$. Thus, condition 5 (orig-virt) remains satisfied.

Condition 6 (subgraph) could only be violated if the subgraph of $T_{\mathcal{S}'}$ formed by the allocation skeletons of some vertex $z \in G_{\mathcal{S}'}$ was no longer connected. This could only happen if only one of G_α and G_β were an allocation skeleton of z, while the other has a further neighbor that is also an allocation skeleton of z. Assume without loss of generality that G_α and the neighbor G_ν of G_β, but not G_β itself, were allocation skeletons of z. Because G_ν and G_β are adjacent in $T_{\mathcal{S}'}$ there are virtual edges $xy = \text{twinE}'(x'y')$ with $xy \in G_\beta$ and $x'y' \in G_\nu$. The same virtual edges are also present in the input instance, only with the difference that $xy \in G_\mu$ and μ (instead of β) and ν are adjacent in $T_\mathcal{S}$. As the input instance satisfies condition 5 (orig-virt), it is $z \in \text{origV}(V_\nu) \cap \text{origV}(V_\mu) = \text{origV}(\{x, y\}) = \text{origV}(\{x', y'\})$. As $\text{origV}(\{x, y\}) = \text{origV}'(\{x, y\})$, this is a contradiction to G_β not being an allocation skeleton of z.

Finally, the mapping origE remains unchanged and the only change to origV is to include two new vertices mapping to already existing vertices. Due to condition 4 (orig-real) holding for both the input and the output instance, this cannot affect the represented graph $G_{\mathcal{S}'}$.

Now consider the skeleton decomposition $\mathcal{S}'$ returned by JoinSeparationPair. Identifying distinct vertices of distinct connected components does not affect their biconnectivity, thus condition 1 (bicon) remains satisfied. The operation effectively contracts and removes an edge in $T_\mathcal{S}$, which does not affect $T_{\mathcal{S}'}$ being a tree satisfying condition 2 (tree). Note that condition 2 (tree) holding for the input instance also ensures that G_α and G_β are two distinct skeletons. As the input instance also satisfies condition 5 (orig-virt), there are exactly two vertices in each of the two adjacent skeletons G_α and G_β, where origV maps to the same vertex of $G_\mathcal{S}$. These two vertices must be part of the twinE pair making the two skeletons adjacent, thus they are exactly the two pairs of vertices we identify with each other. Thus, $\text{origV}|_{V_\mu}$ is still injective, satisfying condition 3 (orig-inj). As we modify no real edges and no other virtual edges, the mappings origV′ and origE′ obviously still satisfy condition 4 (orig-real). As the allocation skeletons of each graph vertex form a connected

subgraph, joining two skeletons cannot change the intersection with any of their neighbors, leaving 5 (orig-virt) satisfied. Finally, contracting a tree edge cannot lead to any of the subgraphs of 6 (subgraph) becoming disconnected, thus the condition also remains satisfied. Again, no changes were made to origE, while condition 5 (orig-virt) makes sure that origV mapped the two pairs of merged vertices to the same vertex of $G_{\mathcal{S}}$. Thus, the represented graph $G_{\mathcal{S}'}$ remains unchanged. ■

This gives us a second way of finding the represented graph by exhaustively joining all skeletons until there is only one left, obtaining the unique trivial skeleton decomposition:

▶ **Lemma 7.2.** Exhaustively applying `JoinSeparationPair` to a skeleton decomposition $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE})$ yields a trivial skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$ where $T_{\mathcal{S}'}$ consists of a single node μ and origE′ and origV′ define an isomorphism between G'_μ and $G_{\mathcal{S}'}$. ◀

Proof. As all virtual edges are matched, and the matched virtual edge always belongs to a different skeleton (condition 2 (tree) ensures that $T_{\mathcal{S}}$ is loop-free), we can always apply `JoinSeparationPair` on a virtual edge until there are none left. As $T_{\mathcal{S}}$ is connected, this means that the we always obtain a tree with a single node, that is an instance with only a single skeleton. As a single application of `JoinSeparationPair` preserves the represented graph, any chain of multiple applications also does. Note that origE′ is a bijection and the surjective origV′ is also injective on the single remaining skeleton due to condition 3 (orig-inj), thus it also globally is a bijection. Together with condition 4 (orig-real), this ensures that any two vertices u and v of G'_μ are adjacent if and only if $\text{origV}'(u)$ and $\text{origV}'(v)$ are adjacent in $G_{\mathcal{S}'}$. Thus origV′ is an edge-preserving bijection, that is an isomorphism. ■

A key point about the skeleton decomposition and especially the operation `SplitSeparationPair` is that they model the decomposition of a graph at separation pairs. This decomposition was formalized as *SPQR-tree* by Di Battista and Tamassia [DT96a; DT96b] and is unique for a given graph [HT73b; Mac37]. Angelini et al. [ABR14] describe a decomposition tree that is conceptually equivalent to our skeleton decomposition. They also present an alternative definition for the SPQR-tree as a decomposition tree satisfying further properties. We adopt this definition as follows, not requiring planarity of triconnected components and allowing virtual edges and real edges to appear within one skeleton (i.e., having leaf Q-nodes merged into their parents).

▶ **Definition 7.3.** A skeleton decomposition $\mathcal{S}$ where any skeleton in $\mathcal{G}$ is either a polygon, a bond, or triconnected (“rigid”), and two skeletons adjacent in $T_{\mathcal{S}}$ are never both polygons or both bonds, is the unique SPQR-tree of $G_{\mathcal{S}}$. ◀

The main difference between the well-known ideas behind decomposition trees and our skeleton decomposition is that the latter allow an axiomatic access to the decomposition at separation pairs. For the skeleton decomposition, we employ a purely functional, "mechanical" data structure instead of relying on and working along a given graph structure. In our case, the represented graph is deduced from the data structure (i.e. the SPQR-tree) instead of computing the data structure from the graph.

7.2 Extended Skeleton Decompositions

Note that most skeletons, especially polygons and bonds, can easily be decomposed into smaller parts. The only exception to this are triconnected skeletons which cannot be split further using the operations we defined up to now. This is a problem when modifying a vertex that occurs in triconnected skeletons that may be much bigger than the direct neighborhood of the vertex. To fix this, we define a further set of operations that allow us to isolate vertices out of arbitrary triconnected components by replacing them with a ("virtual") placeholder vertex. This placeholder then points to a smaller component that contains the actual vertex, see Figure 7.3. Modification of the edges incident to the placeholder is disallowed, which is why we call them "occupied".

Formally, the structures needed to keep track of the components split in this way in an *extended* skeleton decomposition $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE}, \text{twinV})$ are defined as follows. Skeletons now have the form $G_\mu = (V_\mu \cup\!\!\!\cdot\, V_\mu^{\text{virt}}, E_\mu^{\text{real}} \cup\!\!\!\cdot\, E_\mu^{\text{virt}} \cup\!\!\!\cdot\, E_\mu^{\text{occ}})$. Bijection $\text{twinV} : V^{\text{virt}} \to V^{\text{virt}}$ matches all *virtual vertices* $V^{\text{virt}} = \bigcup_\mu V_\mu^{\text{virt}}$, such that $\text{twinV}(v) \neq v$, $\text{twinV}^2 = \text{id}$. The edges incident to virtual vertices are contained in E_μ^{occ} and thus considered *occupied*; see Figure 7.3 (b). Similar to the virtual edges matched by twinE, any two virtual vertices matched by twinV induce an edge between their skeletons in $T_\mathcal{S}$. Condition 2 (tree) also equally applies to those edges induced by twinV, which in particular ensures that there are no parallel twinE and twinV tree edges in $T_\mathcal{S}$. Similarly, the connected subgraphs of condition 6 (subgraph) can also contain tree edges induced by twinV. All other conditions remain unchanged, but we add two further conditions to ensure that twinV is consistent:

7 (stars) For each v_α, v_β with $\text{twinV}(v_\alpha) = v_\beta$, it is $\deg(v_\alpha) = \deg(v_\beta)$. All edges incident to v_α and v_β are occupied and have distinct endpoints (except for v_α and v_β). Each occupied edge is adjacent to exactly one virtual vertex.

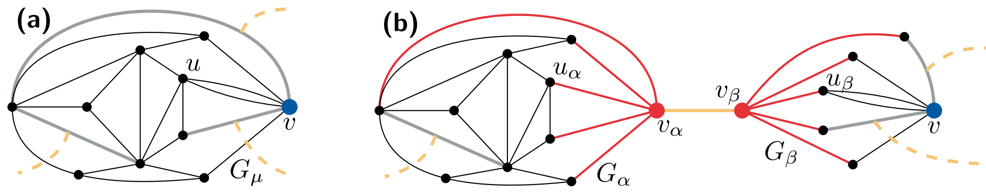


Figure 7.3: **(a)** A triconnected skeleton G_μ with a highlighted vertex v incident to two gray virtual edges. **(b)** The result of applying `IsolateVertex` to isolate v out of the skeleton. The red occupied edges in the old skeleton G_α form a star with center v_α, while the red occupied edges in G_β connect all neighbors of v to form a star with center $v_\beta \neq v$. The centers v_α and v_β are virtual and matched with each other. Neighbor u of v was split into vertices u_α and u_β.

8 (orig-stars) Let v_α and v_β again be two virtual vertices matched with each other by twinV. For their respective skeletons G_α and G_β (where α and β are adjacent in $T_\mathcal{S}$), we have $\text{origV}(V_\alpha) \cap \text{origV}(V_\beta) = \text{origV}(N(v_\alpha)) = \text{origV}(N(v_\beta))$.

Both conditions together yield a bijection $\gamma_{v_\alpha v_\beta}$ between the neighbors of v_α and the neighbors of v_β, as origV is injective when restricted to a single skeleton (condition 3 (orig-inj)) and $\deg(v_\alpha) = \deg(v_\beta)$. Operations `SplitSeparationPair` and `JoinSeparationPair` can also be applied to an extended skeleton decomposition, yielding an extended skeleton decomposition without modifying twinV. To ensure that conditions 7 (stars) and 8 (orig-stars) remain unaffected by both operations, `SplitSeparationPair` can only be applied to non-virtual vertices.

The operations `IsolateVertex` and `Integrate` now allow us to isolate vertices out of triconnected components and integrate them back in, respectively. For `IsolateVertex`, let v be a non-virtual vertex of skeleton G_μ, such that v has no incident occupied edges. Applying `IsolateVertex`$(\mathcal{S}, v)$ on an extended skeleton decomposition $\mathcal{S}$ yields an extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ as follows. Each neighbor u of v is split into two non-adjacent vertices u_α and u_β, where u_β is incident to all edges connecting u with v, while u_α keeps all other edges of u. We set $\text{origV}'(u_\alpha) = \text{origV}'(u_\beta) = \text{origV}(u)$. This creates an independent, star-shaped component with center v, which we move to skeleton G_β, while we rename skeleton G_μ to G_α. We connect all u_α to a single new virtual vertex $v_\alpha \in V_\alpha^{\text{virt}}$ using occupied edges, and all u_β to a single new virtual vertex $v_\beta \in V_\beta^{\text{virt}}$ using occupied edges; see Figure 7.3. Finally, we set $\text{twinV}'(v_\alpha) = v_\beta$, $\text{twinV}'(v_\beta) = v_\alpha$, and add G_β to $\mathcal{G}'$. All other mappings and skeletons remain unchanged.

For `Integrate`, consider two virtual vertices v_α, v_β with $\text{twinV}(v_\alpha) = v_\beta$ and the bijection $\gamma_{v_\alpha v_\beta}$ between the neighbors of v_α and v_β. An application of `Integrate`$(\mathcal{S}, (v_\alpha, v_\beta))$ yields an extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ as follows. We merge both skeletons into a skeleton G_μ (also replacing both in $\mathcal{G}'$) by identifying the neighbors of v_α and v_β according to $\gamma_{v_\alpha v_\beta}$. Furthermore, we remove v_α and v_β together with their incident occupied edges. All other mappings and skeletons remain unchanged.

▶ **Lemma 7.4.** Applying `IsolateVertex` or `Integrate` on an extended skeleton decomposition $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE}, \text{twinV})$ yields an extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ with $G_{\mathcal{S}'} = G_{\mathcal{S}}$. ◀

Proof. We first check that all conditions still hold in the extended skeleton decomposition $\mathcal{S}'$ returned by `IsolateVertex`. Condition 1 (bicon) remains satisfied, as the structure of G_α remains unchanged compared to G_μ and the skeleton G_β is a bond. As we are again splitting a node of $T_{\mathcal{S}}$, condition 2 (tree) also remains satisfied. Due to the neighbors of v_β and v_α mapping to the same vertices of $G_{\mathcal{S}'}$, conditions 3 (orig-inj), 4 (orig-real), and 5 (orig-virt) remain satisfied. Conditions 7 (stars) and 8 (orig-stars) are satisfied by construction.

Lastly, condition 6 (subgraph) could only be violated if the subgraph of $T_{\mathcal{S}'}$ formed by the allocation skeletons of some vertex $z \in G_{\mathcal{S}'}$ was no longer connected. This could only happen if only one of G_α and G_β were an allocation skeleton of z, while the other has a further neighbor G_ν that is also an allocation skeleton of z. Note that in any case, ν is adjacent to μ in $T_{\mathcal{S}}$ and μ must be an allocation skeleton of z, thus it is $z \in \text{origV}(G_\nu) \cap \text{origV}(G_\mu)$. Depending on the adjacency of ν, it is either $\text{origV}(G_\nu) \cap \text{origV}(G_\mu) = \text{origV}'(G_\nu) \cap \text{origV}(G_\alpha)$ or $\text{origV}(G_\nu) \cap \text{origV}(G_\mu) = \text{origV}'(G_\nu) \cap \text{origV}(G_\beta)$, as ν is not modified by the operation and both $\mathcal{S}$ and $\mathcal{S}'$ satisfy 5 (orig-virt) and 8 (orig-stars). This immediately contradicts the skeleton of $\{\alpha, \beta\}$, that is adjacent to ν, not being an allocation skeleton of z.

Finally, the mapping origE remains unchanged and the only change to origV is to include some duplicated vertices mapping to already existing vertices. Due to condition 4 (orig-real) holding for both the input and the output instance, this cannot affect the represented graph $G_{\mathcal{S}'}$.

Now consider the extended skeleton decomposition $\mathcal{S}'$ returned by `Integrate`. The merged skeleton is biconnected, as we are effectively replacing a single vertex by a connected subgraph, satisfying 1 (bicon). The operation effectively contracts and removes an edge in $T_{\mathcal{S}}$, which does not affect $T_{\mathcal{S}'}$ being a tree, satisfying condition 2 (tree). Note that condition 2 (tree) holding for the input instance also ensures that v_α and v_β belong to two distinct skeletons. As the input instance satisfies condition 5 (orig-virt), the vertices in each of the two adjacent skeletons where

origV maps to the same vertex of $G_{\mathcal{S}}$ are exactly the neighbors of the matched v_α and v_β. Thus, $\text{origV}\,|_{V_\alpha}$ is still injective, satisfying condition 3 (orig-inj). As we modify no real or virtual edges, the mappings origV′, origE′ and twinE′ obviously still satisfy conditions 4 (orig-real) and 5 (orig-virt). Finally, contracting a tree edge cannot lead to any of the subgraphs of 6 (subgraph) becoming disconnected, thus the condition also remains satisfied. Conditions 7 (stars) and 8 (orig-stars) also remain unaffected, as we simply remove an entry from twinV.

Again, no changes were made to origE, while condition 8 (orig-stars) makes sure that origV mapped each pair of merged vertices to the same vertex of $G_{\mathcal{S}}$. Thus, the represented graph $G_{\mathcal{S}'}$ remains unchanged. ■

Furthermore, as `Integrate` is the converse of `IsolateVertex` and has no preconditions, any changes made by `IsolateVertex` can be undone at any time to obtain a (non-extended) skeleton decomposition, and thus possibly the SPQR-tree of the represented graph.

▶ **Remark 7.5.** Exhaustively applying `Integrate` to an extended skeleton decomposition $\mathcal{S} = (\mathcal{G}, \text{origV}, \text{origE}, \text{twinE}, \text{twinV})$ yields a extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ where $\text{twinV}' = \emptyset$. Thus, $\mathcal{S}'$ is equivalent to a (non-extended) skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}')$. ◀

7.3 Node Expansion in Extended Skeleton Decompositions

We now introduce the dynamic operation that changes the represented graph by expanding a single vertex u into an arbitrary connected graph G_ν. This is done by identifying $|N(u)|$ marked vertices in G_ν with the neighbors of u via a bijection ϕ and then removing u and its incident edges. We use the "occupied stars" from the previous section to model the identification of these vertices, allowing us to defer the actual insertion to an application of `Integrate`. We need to ensure that the inserted graph makes the same "guarantees" to the surrounding graph in terms of connectivity as the vertex it replaces, that is all neighbors of u (i.e. all marked vertices in G_ν) need to be pairwise connected via paths in G_ν, even without using any other neighbor of u (i.e. any other marked vertex). Without this requirement, a single vertex could e.g. also be split into two non-adjacent halves, which could break a triconnected component apart. Thus, we require G_ν to be biconnected when all marked vertices are collapsed into a single vertex. Note that this also ensures

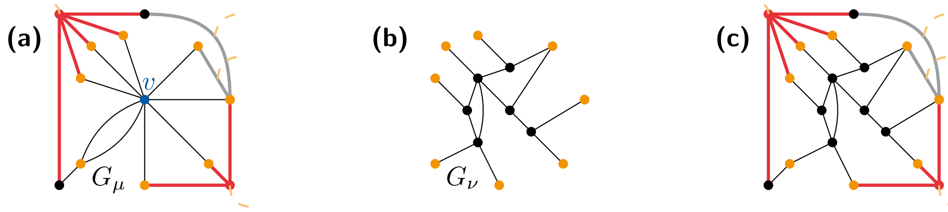


Figure 7.4: Expanding a skeleton vertex v into a graph G_ν in the SPQR-tree of Figure 7.5 (b). **(a)** The single allocation skeleton G_μ of u with the single allocation vertex v of u from Figure 7.5 (b). The neighbors of v are marked in orange. **(b)** The inserted graph G_ν with orange marked vertices. Note that the graph is biconnected when all marked vertices are collapsed into a single vertex. **(c)** The result of applying $\texttt{InsertGraph}(\mathcal{S}, u, G_\nu, \phi)$ followed by an application of $\texttt{Integrate}$ on the generated virtual vertices v and v'.

that the old graph can be restored by contracting the vertices of the inserted graph. For the sake of simplicity, we require vertex u from the represented graph to have a single allocation vertex $v \in G_\mu$ with $\text{origV}^{-1}(u) = \{v\}$ so that we only need to change a single allocation skeleton G_μ in the skeleton decomposition. As we will make clear later on, this condition can be established easily.

Formally, let $u \in G_\mathcal{S}$ be a vertex that only has a single allocation vertex $v \in G_\mu$ (and thus only a single allocation skeleton G_μ). Let G_ν be an arbitrary, new graph containing $|N(u)|$ marked vertices, together with a bijection ϕ between the marked vertices in G_ν and the neighbors of v in G_μ. We require G_ν to be biconnected when all marked vertices are collapsed into a single node. Operation $\texttt{InsertGraph}(\mathcal{S}, u, G_\nu, \phi)$ yields an extended skeleton decomposition $\mathcal{S}' = (\mathcal{G}', \text{origV}', \text{origE}', \text{twinE}', \text{twinV}')$ as follows, see also Figure 7.4. We interpret G_ν as skeleton and add it to $\mathcal{G}'$. For each marked vertex x in G_ν, we set $\text{origV}'(x) = \text{origV}(\phi(x))$. For all other vertices and edges in G_ν, we set origV′ and origE′ to point to new vertices and edges forming a copy of G_ν in $\mathcal{G}_{\mathcal{S}'}$. We connect every marked vertex in G_ν to a new virtual vertex $v' \in G_\nu$ using occupied edges. We also convert v to a virtual vertex, converting its incident edges to occupied edges while removing parallel edges. Finally, we set $\text{twinV}'(v) = v'$ and $\text{twinV}'(v') = v$.

▶ **Lemma 7.6.** Applying $\texttt{InsertGraph}(\mathcal{S}, u, G_\nu, \phi)$ on an extended skeleton decomposition $\mathcal{S}$ yields an extended skeleton decomposition $\mathcal{S}'$ with $G_{\mathcal{S}'}$ isomorphic to $G_\mathcal{S}[u \rightarrow_\phi G_\nu]$. ◀

Proof. Condition 1 (bicon) remains satisfied, as the structure of G_μ remains unchanged and the resulting G_ν is biconnected by precondition. Regarding $T_\mathcal{S}$, we are attaching a degree-1 node ν to an existing node μ, thus condition 2 (tree) also remains satisfied. As all vertices of G_ν except for the vertices in $N(v')$ got their

new, unique copy assigned by origV′ and $\mathrm{origV}'(N(v')) = \mathrm{origV}(N(v))$, condition 3 (orig-inj) is also satisfied for the new G_ν. As we updated origE alongside origV and G_ν contains no virtual edges, conditions 4 (orig-real) and 5 (orig-virt) remain satisfied. As ν is a leaf of $T_\mathcal{S}$ with μ being its only neighbor, $\mathrm{origV}'(N(v')) \subset \mathrm{origV}(V_\mu)$, and G_ν is the only allocation skeleton for all vertices in $G_\nu \setminus N(v')$, condition 6 (subgraph) remains satisfied. Conditions 7 (stars) and 8 (orig-stars) are satisfied by construction. Finally, the mappings origE′ and origV′ are by construction updated to correctly reproduce the structure of G_ν in $G_{\mathcal{S}'}$. ■

On its own, this operation is not of much use though, as graph vertices only rarely have a single allocation skeleton. Furthermore, our goal is to dynamically maintain SPQR-trees, while this operation on its own will in most cases not yield an SPQR-tree. To fix this, we introduce the full procedure $\mathtt{InsertGraph}_{\mathtt{SPQR}}(\mathcal{S}, u, G_\nu, \phi)$ that can be applied to any graph vertex u and that, given an SPQR-tree $\mathcal{S}$, yields the SPQR-tree of $G_\mathcal{S}[u \to_\phi G_\nu]$. It consists of three preparations steps, the insertion of G_ν, and two further clean-up steps:

1. Apply `SplitSeparationPair` to each polygon allocation skeleton of u with more than three vertices, using the neighbors of the allocation vertex of u as separation pair.

2. For each rigid allocation skeleton of u, move the contained allocation vertex v of u to its own skeleton by applying $\mathtt{IsolateVertex}(\mathcal{S}, v)$.

3. Exhaustively apply `JoinSeparationPair` to any pair of allocation skeletons of u that are adjacent in $T_\mathcal{S}$. Due to condition 6 (subgraph), this yields a single component G_μ that is the sole allocation skeleton of u with the single allocation vertex v of u. Furthermore, the size of G_μ is linear in $\deg(u)$.

4. Apply `InsertGraph` to insert G_ν as skeleton, followed by an application of `Integrate` to the virtual vertices $\{v, v'\}$ introduced by the insertion, thus integrating G_ν into G_μ.

5. Apply `SplitSeparationPair` to all separation pairs in G_μ that do not involve a virtual vertex. These pairs can be found in linear time, e.g. by temporarily duplicating all virtual vertices and their incident edges and then computing the SPQR-tree.[12]

12 Later, in Theorem 7.10, we use wheels to replace virtual vertices, which also ensures this.

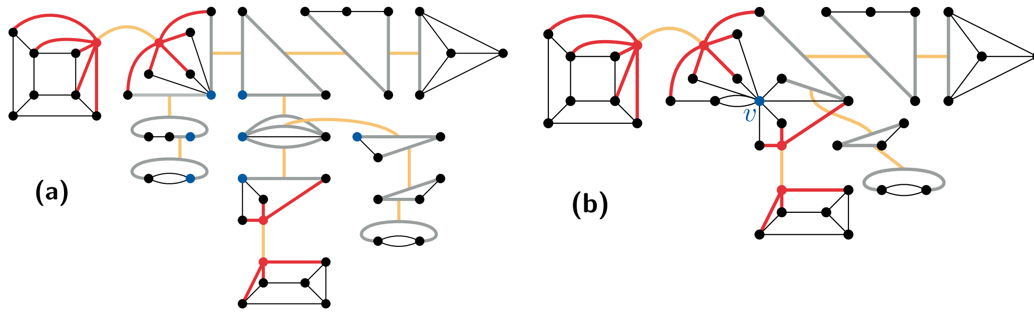


Figure 7.5: The preprocessing steps of $\texttt{InsertGraph}_{\texttt{SPQR}}$ being applied to the SPQR-tree of Figure 7.1 (b). **(a)** The state after step 2, after all allocation skeletons of u have been split. **(b)** The state after step 3, after all allocation skeletons of u have been merged into a single one.

6. Finally, exhaustively apply `Integrate` and also apply `JoinSeparationPair` to any two adjacent polygons and to any two adjacent bonds to obtain the SPQR-tree of the updated graph.

The basic idea behind the correctness of this procedure is that splitting the newly inserted component according to its SPQR-tree in step 5 yields biconnected components that are each either a polygon, a bond, or "almost" triconnected. The latter (and only those) might still contain virtual vertices and all their remaining separation pairs, which were not split in step 5, contain one of these virtual vertices. This, together with the fact that there still may be pairs of adjacent skeletons where both are polygons or both are bonds, prevents the instance from being an SPQR-tree. Both issues are resolved in step 6: The adjacent skeletons are obviously fixed by the `JoinSeparationPair` applications. To show that the removal of virtual vertices by the `Integrate` applications makes the remaining components triconnected, we need the following lemma.

▶ **Lemma 7.7.** Let G_α be a triconnected skeleton containing a virtual vertex v_α matched with a virtual vertex v_β of a biconnected skeleton G_β. Furthermore, let $P \subseteq \binom{V(G_\beta)}{2}$ be the set of all separation pairs in G_β. An application of `Integrate` $(\mathcal{S}, (v_\alpha, v_\beta))$ yields a biconnected skeleton G_μ whose separation pairs are precisely

$$P' = \{\{u, v\} \in P \mid v_\beta \notin \{u, v\}\}.$$ ◀

Proof. We partition the vertices of G_μ into sets A, B, and N depending on whether the vertex stems from G_α, G_β, or both, respectively. The set N thus contains the neighbors of v_α, which were identified with the neighbors of v_β. We will

show by contradiction that G_μ contains no separation pairs except for those in P'. Thus, consider a separation pair $u, v \in G_\mu$ not in P'. First, consider the case where $u, v \in A \cup N$. Observe that removing u, v in this case leaves B connected. Thus, we can contract all vertices of B into a single vertex, reobtain G_α and see that u, v is a separation pair in G_α. This contradicts the precondition that G_α is triconnected. Now consider the case where $u, v \in B \cup N$. Analogously to above, we find that u, v is a separation pair in G_β that does not contain v_β, a contradiction to $\{u, v\} \notin P'$. Finally, consider the remaining case where, without loss of generality, $u \in A, v \in B$. Since $\{u, v\}$ is a separation pair, u has two neighbors x, y that lie in different connected components of $G_\mu - \{u, v\}$ and therefore also in different components of $(G_\mu - \{u, v\}) - B$ which is isomorphic to $G_\alpha - \{u, v_\alpha\}$. This again contradicts G_α being triconnected. ■

▶ **Theorem 7.8.** Applying $\texttt{InsertGraph}_{\texttt{SPQR}}(\mathcal{S}, u, G_\nu, \phi)$ to an SPQR-tree $\mathcal{S}$ yields an SPQR-tree $\mathcal{S}'$ in $O(|G_\nu|)$ time with $G_{\mathcal{S}'}$ isomorphic to $G_{\mathcal{S}}[u \to_\phi G_\nu]$. ◀

Proof. As all applied operations leave the extended skeleton decomposition valid, the final extended skeleton decomposition $\mathcal{S}'$ is also valid. The purpose of the preprocessing steps 1 to 3 is to ensure that the preconditions of `InsertGraph` are satisfied and the affected component is not too large. All rigids split in step 2 remain structurally unmodified in the sense that edges only changed their type, but the graph and especially its triconnectedness remains unchanged. step 4 performs the actual insertion and yields the desired represented graph according to Lemma 7.6. It thus remains to show that the clean-up steps turn the obtained extended skeleton decomposition into an SPQR-tree. Applying `Integrate` exhaustively in step 6 ensures that the extended skeleton decomposition is equivalent to a non-extended one (Remark 7.5). Recall that a non-extended skeleton decomposition is an SPQR-tree if all skeletons are either polygons, bonds or triconnected and two adjacent skeletons are never both polygons or both bonds (Definition 7.3). step 6 ensures that the second half holds, as joining two polygons (or two bonds) with `JoinSeparationPair` yields a bigger polygon (or bond, respectively). Before step 6, all skeletons that are not an allocation skeleton of u are still unmodified and thus already have a suitable structure, i.e., they are either polygons, bonds or triconnected. Furthermore, the allocation skeletons of u not containing virtual vertices also have a suitable structure, as their splits were made according to the SPQR-tree in step 5. It remains to show that the remaining skeletons, that is those resulting from the `Integrate` applications in step 6, are triconnected. Note that in these skeletons, step 5 ensures that every separation pair consists of at least one virtual vertex, as otherwise the computed SPQR-tree would have split the skeleton. Further note that, for each of these virtual vertices, the matched partner vertex

is part of a structurally unmodified triconnected skeleton that was split in step 2. Lemma 7.7 shows that applying `Integrate` does not introduce new separation pairs while removing two virtual vertices if one of the two sides is triconnected. We can thus exhaustively apply `Integrate` and thereby remove all virtual vertices and all separation pairs, obtaining triconnected components. This shows that the criteria for being an SPQR-tree are satisfied and, as `InsertGraph` expanded u to G_v in the represented graph, we now have the unique SPQR-tree of $G_{\mathcal{S}}[u \rightarrow_\phi G_v]$.

All operations we used can be performed in time linear in the degree of the vertices they are applied on. For the bipartition of bridges that is the input to `SplitSeparationPair`, it is sufficient to describe each bridge via its edges incident to the separation pair instead of explicitly enumerating all vertices in the bridge. Thus, the applications of `SplitSeparationPair` and `IsolateVertex` in steps 1 and 2 touch every edge incident to u at most once and thus take $O(\deg(u))$ time. Furthermore, they yield skeletons that have a size linear in the degree of their respective allocation vertex of u. As the subtree of u's allocation skeletons has size at most $\deg(u)$, the `JoinSeparationPair` applications of step 3 also take at most $O(\deg(u))$ time. It follows that the resulting single allocation skeleton of u has size $O(\deg(u))$. The applications of `InsertGraph` and `Integrate` in step 4 take time linear in the number of identified neighbors, which is $O(\deg(u))$. Generating the SPQR-tree of the inserted graph in step 5 (where all virtual vertices where replaced by wheels) can be done in time linear in the size of the inserted graph [GM00; HT73b], that is $O(|G_v|)$. Applying `SplitSeparationPair` according to all separation pairs identified by this SPQR-tree can also be done in $O(|G_v|)$ time in total. Note that there are at most $\deg(u)$ edges between the skeletons that existed before step 4 and those that were created or modified in steps 4 and 5, and these are the only edges that might now connect two polygons or two bonds. As these tree edges have one endpoint in the single allocation skeleton of u, the applications of `Integrate` and `JoinSeparationPair` in step 6 run in $O(\deg(u))$ time in total. Furthermore, they remove all pairs of adjacent polygons and all pairs of adjacent bonds. This shows that all steps take $O(\deg(u))$ time – except for step 5, which takes $O(|G_v|)$ time. As the inserted graph contains at least one vertex for each neighbor of u, the total running time is in $O(|G_v|)$. ■

▶ **Corollary 7.9.** Let $\mathcal{S}_1, \mathcal{S}_2$ be two SPQR-trees together with vertices $u_1 \in G_{\mathcal{S}_1}$, $u_2 \in G_{\mathcal{S}_2}$, and let ϕ be a bijection between the edges incident to u_1 and the edges incident to u_2. Operation $\texttt{Merge}_{\texttt{SPQR}}(\mathcal{S}_1, \mathcal{S}_2, u_1, u_2, \phi)$ yields the SPQR-tree of the graph $G_{\mathcal{S}_1}[u_1 \rightarrow_\phi (G_{\mathcal{S}_2} - u_2)]$, i.e. the union of both graphs where the edges incident to u_1, u_2 were identified according to ϕ and u_1, u_2 removed, in time $O(\deg(u_1)) = O(\deg(u_2))$. ◀

Proof. Operation $\texttt{Merge}_{\text{SPQR}}$ works similarly to the more general $\texttt{InsertGraph}_{\text{SPQR}}$, although the running time is better because we already know the SPQR-tree for the graph being inserted. We apply steps 1 to 3 on both decompositions to ensure that both u_1 and u_2 have sole allocation vertices v_1 and v_2, respectively. To properly handle parallel edges, we subdivide all edges incident to u_1, u_2 (and thus also the corresponding real edges incident to v_1, v_2) and then identify the subdivision vertices of each pair of edges matched by ϕ. By deleting vertices v_1 and v_2 and suppressing the subdivision vertices (that is, removing them and identifying each pair of incident edges) we obtain a skeleton G_μ that has size $O(\deg(u_1)) = O(\deg(u_2))$. Finally, we apply steps 5 and 6 to G_μ to obtain the final SPQR-tree. Again, as the partner vertex of every virtual vertex in the allocation skeletons of u is part of a triconnected skeleton, applying $\texttt{Integrate}$ exhaustively in step 6 yields triconnected skeletons. As previously discussed, the preprocessing and clean-up steps run in time linear in degree of the affected vertices, thus the overall running time is $O(\deg(u_1)) = O(\deg(u_2))$ in this case. ■

7.3.1 Maintaining Planarity and Vertex Rotations

Expanding a vertex of a planar graph using another planar graph using $\texttt{InsertGraph}_{\text{SPQR}}$ (or merging two SPQR-trees of planar graphs using Corollary 7.9) can yield a non-planar graph. This is, e.g., because the rigids of both graphs might require incompatible orders for the neighbors of the replaced vertex. The aim of this section is to efficiently detect this case, that is a planar graph turning non-planar by an expansion. To check a general graph for planarity, it suffices to check the rigids in its SPQR-tree for planarity [DT96b]. Thus, if a graph becomes non-planar through an application of $\texttt{InsertGraph}_{\text{SPQR}}$, this will be noticeable from the triconnected allocation skeletons of the replaced vertex. To be able to immediately report if the instance became non-planar, we need to maintain a rotation, that is a cyclic order of all incident edges, for each vertex in a triconnected skeleton. We do not track the direction of the orders, that is we only store the order up to reversal. As discussed later, the exact orders can also be maintained with a slight overhead.

▶ **Theorem 7.10.** SPQR-trees support the following operations:

- $\texttt{InsertGraph}_{\text{SPQR}}(\mathcal{S}, u, G_\nu, \phi)$: expansion of a single vertex u in time $O(|G_\nu|)$,
- $\texttt{Merge}_{\text{SPQR}}(\mathcal{S}_1, \mathcal{S}_2, u_1, u_2, \phi)$: merging of two SPQR-trees in time $O(\deg(u_1))$,
- $\texttt{IsPlanar}$: queries whether the represented graph is planar in time $O(1)$, and
- $\texttt{Rotation}(u)$: queries for one of the two possible rotations of vertices u in planar triconnected skeletons in time $O(1)$. ◀

Proof. Note that the flag IsPlanar together with the Rotation information can be computed in linear time when creating a new SPQR-tree and that expanding a vertex or merging two SPQR-trees cannot turn a non-planar graph planar. We make the following changes to the operations $\mathtt{InsertGraph}_{\mathrm{SPQR}}$ and $\mathtt{Merge}_{\mathrm{SPQR}}$ to maintain the new information. After a triconnected component is split in step 2 we now introduce further structure to ensure that the embedding is maintained on both sides. The occupied edges generated around the split-off vertex v (and those around its copy v') are subdivided and connected cyclically according to $\mathtt{Rotation}(v)$. Instead of "stars", we thus now generate occupied "wheels" that encode the edge orders in the embedding of the triconnected component. When generating the SPQR-tree of the modified subgraph in step 5, we also generate a planar embedding for all its triconnected skeletons. If no planar embedding can be found for at least one skeleton, we report that the resulting instance is non-planar by setting IsPlanar to false. Otherwise, after performing all splits indicated by the SPQR-tree, we assign Rotation by generating embeddings for all new rigids. Note that for all skeletons with virtual vertices, the generated embedding will be compatible with the one of the neighboring triconnected component, that is, the rotation of each virtual vertex will line up with that of its matched partner vertex, thanks to the inserted wheel. Finally, before applying Integrate in step 6, we contract each occupied wheel into a single vertex to re-obtain occupied stars. The creation and contraction of wheels adds an overhead that is at most linear in the degree of the expanded vertex and the generation of embeddings for the rigids can be done in time linear in the size of the rigid. Thus, this does not affect the asymptotic running time of both operations. ■

► **Corollary 7.11.** The data structure from Theorem 7.10 can be adapted to also provide the exact rotations with matching direction for every vertex in a rigid. Furthermore, it can support queries whether two vertices v_1, v_2 are connected by at least three vertex-disjoint paths via $\mathtt{3Paths}(v_1, v_2)$ in $O((\deg(v_1) + \deg(v_2)) \cdot \alpha(n))$ time, where α is the inverse Ackermann function. These adaptions change the running time of $\mathtt{InsertGraph}_{\mathrm{SPQR}}$ to $O(\deg(u) \cdot \alpha(n) + |G_v|)$, that of $\mathtt{Merge}_{\mathrm{SPQR}}$ to $O(\deg(u_1) \cdot \alpha(n))$, and that of $\mathtt{Rotation}(u)$ to $O(\alpha(n))$. ◄

Proof. The exact rotation information for Rotation can be maintained by using Union-Find [TL84] to keep track of the rigid a vertex belongs to and synchronizing the reversal of all vertices within one rigid when two rigids are merged by Integrate as follows. We create a Union-Find set for every vertex in a triconnected component and apply Union to all vertices in the same rigid. Next to the pointer indicating the representative in the Union-Find structure, we store a boolean flag

indicating whether the rotation information for the current vertex is reversed with regard to rotation of its direct representative. To find whether a Rotation needs to be flipped, we accumulate all flags along the path to the actual representative of a vertex by using an exclusive-or. As Rotation(u) thus relies on the Find operation, its amortized running time is $O(\alpha(n))$. When merging two rigids with Integrate, we also perform a Union on their respective representatives (which we need to Find first), making Integrate$(\mathcal{S}, (v_\alpha, v_\beta))$ run in $O(\deg(v_\alpha) + \alpha(n))$. We also compare the Rotation of the replaced vertices and flip the flag stored with the vertex that does not end up as the representative if they do not match. In total, this makes InsertGraph$_{\text{SPQR}}$ run in $O(\deg(u) \cdot \alpha(n) + |G_\nu|)$ time as there can be up to $\deg(u)$ split rigids. Furthermore, Merge$_{\text{SPQR}}$ now runs in $O(\deg(u_1) \cdot \alpha(n))$ time.

Maintaining the information which rigid a skeleton vertex is contained in can then also be used to answer queries whether two arbitrary vertices are connected by three disjoint paths. This is exactly the case if they are part of the same rigid, appear as poles of the same bond or are connected by a virtual edge in a polygon. This can be checked by enumerating all allocation skeletons of both vertices, which can be done in time linear in their degree. As finding each of the skeletons may require a Find call, the total running time for this is in $O((\deg(v_1) + \deg(v_2)) \cdot \alpha(n))$. ■

7.4 Applications

In this section, we show how extended skeleton decompositions and their dynamic operation InsertGraph$_{\text{SPQR}}$ can be used to improve the running time of the algorithm from Chapter 6 for solving Synchronized Planarity and how this transfers to other constrained planarity variants. Recall that in Synchronized Planarity, we are given a matching on some of the vertices of a graph and seek a planar embedding where the rotations of matched vertices coincide under a given bijection. The synchronization constraints resulting from the matching of two vertices, which must have the same degree, are called *pipe*. The algorithm for solving Synchronized Planarity works by removing an arbitrary pipe each step, using one of three operations depending on the graphs around the matched vertices; see the overview in Figure 7.6. Some of these operations require embedding trees, which describe all possible rotations of a single vertex in a planar graph and are used to communicate embedding restrictions between vertices with synchronized rotation. Without our optimizations, computing an embedding tree requires time linear in the size of the concerned biconnected component, that is $O(m)$. Once their embedding trees are available, each of the at most $O(m)$ executed operations runs in time linear in the degree of the pipe it is applied on, that is in $O(\Delta)$ time (see Lemmas 6.4 to 6.6).

Thus, being able to generate these embedding trees without an overhead over the operation that uses them by maintaining the SPQR-trees they can be derived from is our main contribution towards the speed-up of the SYNCHRONIZED PLANARITY algorithm. Our experimental evaluation of the SYNCHRONIZED PLANARITY algorithm also shows that its main bottleneck in practice is the generation of embedding trees (see Chapter 9), indicating that these improvements can also help in achieving a significant practical speed-up.

7.4.1 Embedding Trees

Recall that the embedding tree $\mathcal{T}_v$ for vertex v of a biconnected graph G is a PC-tree that describes the possible cyclic orders or rotations of the edges incident to v in all planar embeddings of G [BL76; BR16b]; see Figure 7.1 (d). To generate the embedding tree we use the observation about the relationship of SPQR-trees and embedding trees described by Bläsius and Rutter [BR16b, Section 2.5]: There is a bijection between the P- and C-nodes in the embedding tree of v and the bond and triconnected allocation skeletons of v in the SPQR-tree of G, respectively.

▶ **Lemma 7.12.** Let $\mathcal{S}$ be an SPQR-tree with a planar represented graph $G_\mathcal{S}$. The embedding tree for a vertex $v \in G_\mathcal{S}$ can be found in time $O(\deg(v))$. ◀

Proof. We use the rotation information from Theorem 7.10 and furthermore maintain an (arbitrary) allocation vertex for each vertex in $G_\mathcal{S}$. To compute the embedding tree of a vertex v starting at the allocation vertex u of v, we will explore the SPQR-tree by applying the twinE mapping on one of the edges incident to u and then finding the next allocation vertex of v as one endpoint of the obtained edge. If u has degree 2, it is part of a polygon skeleton that does not induce a node in the embedding tree. We thus move on to its neighboring allocation skeletons and will also similarly skip over any other polygon skeleton we encounter. If u has degree 3 or greater, we inspect two arbitrary incident edges: if they lead to the same vertex, u is the pole of a bond, and we generate a P-node. Otherwise it is part of a triconnected component, and we generate a C-node. We now iterate over the edges incident to u, in the case of a triconnected component using the order given by the rotation of u. For each real edge, we attach a corresponding leaf to the newly generated node. The graph edge corresponding to the leaf can be obtained from the origE mapping. For each virtual edge, we recurse on the respective neighboring skeleton and attach the recursively generated node to the current node. As u can only be part of $\deg(u)$ many skeletons, which form a subtree

of T_S, and the allocation vertices of u in total only have $O(\deg(u))$ many virtual and real edges incident, this procedure yields the embedding tree of u in time linear in its degree. ■

7.4.2 Synchronized Planarity

We now show how we reduce the running time of solving Synchronized Planarity. We do so by generating an SPQR-tree upfront, maintaining it throughout all applied operations, and deriving any needed embedding tree from the SPQR-tree.

▶ **Theorem 7.13.** Synchronized Planarity can be solved in time $O(m \cdot \Delta)$, where m is the number of edges and Δ is the maximum degree of a pipe. ◀

Proof. Recall that the algorithm from Chapter 6 works by splitting (i.e., removing and replacing by smaller ones) the pipes representing synchronization constraints until they are small enough to be trivial. It does so by exhaustively applying the three operations `EncapsulateAndJoin`, `PropagatePQ` and `SimplifyMatching` depending on the graph structure around the pairs of synchronized vertices. As mentioned in the proof of Theorem 6.12, all operations run in time linear in the degree of the pipe they are applied on if the used embedding trees are known, and $O(m)$ operations are sufficient to solve a given instance. Our modification is that we maintain an SPQR-tree for each biconnected component and then generate the needed embedding trees on-demand using Lemma 7.12.

Operation `EncapsulateAndJoin` generates a new bipartite component representing how the edges of the blocks incident to two synchronized cut-vertices are matched with each other. The size of this component is linear in the degree of the synchronized vertices. Thus, we can freshly compute the SPQR-tree for the generated component in linear time, which also does not negatively impact the running time. The only other change made by this operation is that both cut-vertices are split up according to their incident blocks; see Figure 7.6 (a). As this does not affect the SPQR-trees of the blocks, there are no further updates necessary.

`PropagatePQ` takes the non-trivial embedding tree of one synchronized vertex v and inserts copies of the tree in place of v and its partner, respectively. Synchronization constraints on the inner vertices of the inserted trees are used to ensure that the trees are embedded in the same way; see Figure 7.6 (b). We use $\texttt{InsertGraph}_{\text{SPQR}}$ to also insert the embedding tree into the respective SPQR trees, representing C-nodes using wheels. When propagating into a cut-vertex we also need to check whether two or more incident blocks merge. We form equivalence classes on the incident blocks, where two blocks are in the same class if 1) the two subtrees induced by their respective edges share at least two nodes, or 2) both induced

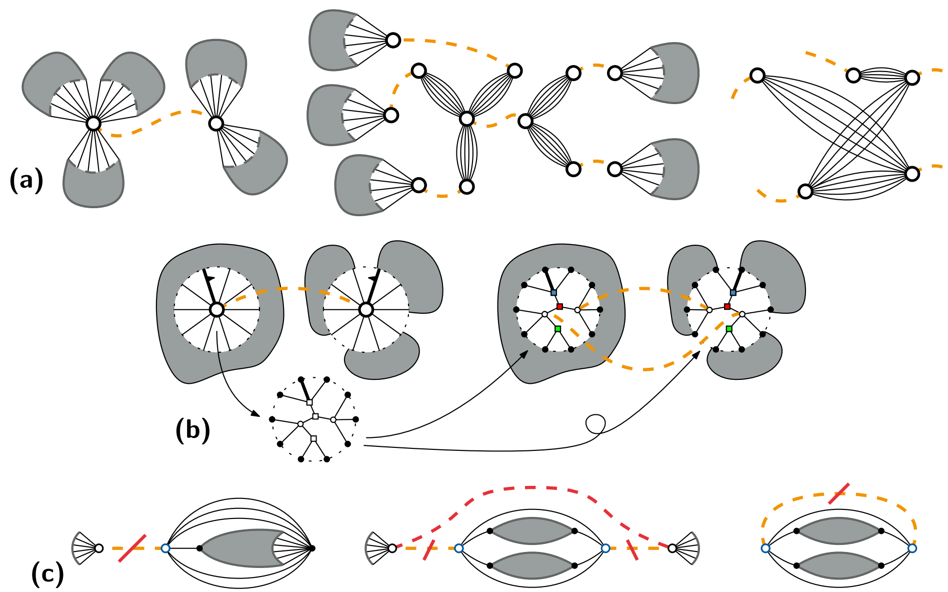


Figure 7.6: Schematic representation of the three operations from Chapter 6 used for solving Synchronized Planarity. Matched vertices are shown as bigger disks, the matching (i.e., the pipes) is indicated by the orange dotted lines. **(a)** Two cut-vertices matched with each other (left), the result of splitting off ("encapsulating") their incident blocks (middle) and the bipartite graph resulting from joining both cut-vertices (right). **(b)** A matched non-cut-vertex with a non-trivial embedding tree (left) that is propagated to replace both the vertex and its partner (right). Constraints that only synchronize a binary decision (e.g. because they correspond to a C-node in the embedding tree) are shown as same-colored squares. **(c)** Three different cases of matched vertices with trivial embedding trees (blue) and how their pipes can be removed or replaced (red).

subtrees share a C-node that has degree at least 2 in both subtrees. Blocks in the same equivalence class will end up in the same biconnected component as follows: We construct the subtree induced by all edges in the equivalence class and add a single further node for each block in the class, connecting all leaves to the node of the block the edges they represent lead to. We compute the SPQR-tree for this biconnected graph and then merge the SPQR-trees of the individual blocks into it by applying Corollary 7.9. As $\texttt{InsertGraph}_{\mathrm{SPQR}}$ (and similarly all $\texttt{Merge}_{\mathrm{SPQR}}$ applications) runs in time linear in the size of the inserted embedding tree, which is asymptotically bounded by the degree of the vertex it represents, this does not negatively impact the running time of the operation.

Operation SimplifyMatching can be applied if the graph around a synchronized vertex v allows arbitrary rotations of v, that is if the embedding tree of v is trivial

(which we can check in $O(\deg(v))$ time). In this case, the pipe can be removed without modifying the graph structure; see Figure 7.6 (c). As this operation makes no changes to the graph, no updates to the SPQR-trees are necessary. ■

Furthermore, as we now no longer need to iterate over whole biconnected components to generate the embedding trees, we are also no longer required to ensure those components do not grow to big. We can thus also directly contract pipes between two distinct biconnected components using Corollary 7.9 instead of having to insert embedding trees using PropagatePQ. This may improve the practical running time, as PropagatePQ might require further operations to clean up the generated pipes, while the direct contraction entirely removes a pipe without generating new ones.

7.4.3 Other Constrained Planarity Variants

The speed-ups for Connected SEFE-2, Partially PQ-constrained Planarity, Row-Column Independent NodeTrix Planarity, and Strip Planarity in Table 7.1 follow directly by combining their linear reductions to Synchronized Planarity from Section 6.4 with the improved running time for Synchronized Planarity from Theorem 7.13. For Clustered Planarity, we provide a more detailed analysis.

▶ **Corollary 7.14.** Clustered Planarity can be solved in time in $O(n + d \cdot \Delta)$, where d is the total number of crossings between cluster borders and edges and Δ is the maximum number of edge crossings on a single cluster border. ◀

Proof. Note that in a planar graph without multi-edges, the number of edges is linear in the number of vertices. We apply the reduction from Clustered Planarity to Synchronized Planarity as described in Section 6.4.2. We then generate an SPQR-tree for every component of the obtained instance with size in $O(n + d)$ in linear time. The instance contains one pipe for every cluster boundary, where the degree of a pipe corresponds to the number of edges crossing the respective cluster boundary. Thus, the potential described in Section 6.3.7, which sums up the degrees of all pipes with a constant factor depending on the endpoints of each pipe, is in $O(d)$. Each operation applied when solving the Synchronized Planarity instance runs in time $O(\Delta)$ (the maximum degree of a pipe) and reduces the potential by at least 1. Thus, a reduced instance without pipes, which can be solved in linear time, can be reached in $O(d \cdot \Delta)$ time. ■

7.5 Conclusion

In this chapter, we have shown how to dynamically maintain an SPQR-tree while expanding vertices into arbitrary biconnected graphs in time linear in the size of the inserted graphs. We also showed how to efficiently merge two SPQR-trees when identifying the edges incident to two vertices with each other. We did this working along an axiomatic definition lifting the SPQR-tree to a stand-alone data structure that can be modified independently from the graph it might have been derived from. Making changes to this structure, we can now observe how the graph represented by the SPQR-tree changes, instead of having to reason which updates to the SPQR-tree are necessary after a change to the represented graph.

Using efficient expansions and merges allowed us to improve the running time of the Synchronized Planarity algorithm from $O(m^2)$ to $O(m \cdot \Delta)$, where Δ is the maximum pipe degree. This also reduced the time for solving several related constrained planarity problems, e.g. for Clustered Planarity from $O((n+d)^2)$ to $O(n + d \cdot \Delta)$, where d is the total number of crossings between cluster borders and edges and Δ is the maximum number of edge crossings on a single cluster border. Our experimental evaluation of the Synchronized Planarity algorithm also shows that its main bottleneck in practice is the generation of embedding trees (see Chapter 9), indicating that these improvements can also help in achieving a significant practical speed-up.

In future work, one may investigate whether `IsolateVertex` could be generalized to not only remove single vertices, but also split triconnected components "in the middle", that is following an edge-cut. This could allow for the reverse operation of the expansion, that is efficient contraction of individual subgraphs, although special care needs to be taken as the contraction might affect the triconnectivity of rigids. A further question we leave for future work is whether our data structure can be combined with that of Holm and Rotenberg [HR20a; HR20b] to allow for efficient edge and vertex insertion as well as deletion together with subgraph expansion.

Part II

Constrained Planarity in Practice

8 Experimental Comparison of PC-Trees and PQ-Trees

This chapter is based on joint work with Matthias Pfretzschner and Ignaz Rutter, which appeared at ESA 2021 [4] where it received the "Best Paper Award". It also appeared in the ACM Journal of Experimental Algorithmics [2], distinguished with the "Reproducible Computation Mark". Our source code is available at github.com/N-Coder/pc-tree and also archived at softwareheritage.org as swh:1:snp:47991f983dd72e5d71486774696f5e02493c3807.

PQ-trees represent linear orders of a ground set subject to constraints that require specific subsets of elements to be consecutive. Similarly, PC-trees do the same for cyclic orders subject to consecutivity constraints. PQ-trees were developed by Booth and Lueker [BL76] to solve the consecutive ones problem, which asks whether the columns of a Boolean matrix can be permuted such that the 1s in each row are consecutive. PC-trees are a more recent generalization introduced by Shih and Hsu [SH99] to solve the circular consecutive ones problem, where the 1s in each row only have to be circularly consecutive.

Though PQ-trees represent linear orders and PC-trees represent cyclic orders, Haeupler and Tarjan [HT08] show that in fact PC-trees and PQ-trees are equivalent, i.e., one can use one of them to implement the other without affecting the asymptotic running time. The main difference between PQ-trees and PC-trees lies in the update procedure. The update procedure takes as input a PQ-tree (a PC-tree) T and a subset U of its leaves and produces a new PQ-tree (PC-tree) T' that represents exactly the linear orders (cyclic orders) represented by T where the leaves in U appear consecutively. The update procedure for PC-trees consists only of a single operation that is applied independently of the structure of the tree. By contrast, the update of the PQ-tree is described in terms of a set of nine template transformations that have to be recursively matched and applied [BL76].

PQ-trees have numerous applications, e.g., in planarity testing [BL76; SH99], recognition of interval graphs [BL76] and genome sequencing [Ben59]. PC-trees have been adopted more widely, e.g., for constrained planarity testing problems [BR16b; BR17; Brü21] due to their simpler update procedure. Despite their wide applications and frequent use in theoretical algorithms, few PQ-tree implementations and even fewer PC-tree implementations are available. Table 8.1 in Section 8.3 shows an overview of all PC- and PQ-tree implementations that we are aware of, though not all of them are working.

In this chapter we describe the first correct and generic implementations of PC-trees. We refer to Section 4.1 for an introduction to PC-trees. Section 8.1 in this chapter contains a more detailed, practice-oriented overview of the update procedure for applying a new restriction to a PC-tree. In Section 8.2, we describe the main challenge when implementing PC-trees and how our two implementations take different approaches at solving it. In Section 8.3, we present an extensive experimental evaluation, where we compare the performance of our implementations with the implementations of PC-trees and PQ-trees from Table 8.1. Our experiments show that PC-trees following Hsu and McConnell's original approach beat their closest competitor, the PQ-tree implementation from the OGDF library [Chi+14] by roughly a factor 2 in terms of running time. Our second implementation using Union-Find is another 50 % faster than this first one, thus beating the OGDF implementation by a factor of up to 4. In Section 8.4, we shortly evaluate the effects this has on the planarity testing algorithms which strongly rely on PC-trees.

8.1 The PC-Tree Update

In this section, we will elaborate on what is necessary for implementing the high-level update from Section 4.1 efficiently. While PC-trees have no designated root nodes and are thus conceptually unrooted, in practice they are usually still rooted at an arbitrary inner node or leaf to simplify implementation.

When applying a restriction R to a PC-tree T with leaf set L, let a leaf $x \in L$ be *full* if $x \in R$ and *empty* otherwise. We call an edge *terminal* if the two subtrees separated by the edge both contain at least one empty and at least one full leaf. Exactly the endpoints of all terminal edges need to be "synchronized", that is have their incident edges ordered in a compatible way, to ensure that all full leaves are consecutive. Hsu and McConnell [HM03; HM04] show that restriction R is possible if and only if the terminal edges form a path and all nodes of this path can be flipped so that all full leaves are on one side and all empty leaves are on the other. Recall that this path is called the *terminal path*, the two nodes at the ends of the terminal path are the *terminal nodes*. Figure 8.1 (a) illustrates the terminal path, while Figure 8.1 (b) shows the final result of updating the PC-tree. The changes made by the update are illustrated by Figure 4.2 in Section 4.1.

We now discuss how to efficiently find the terminal edges, and thus the subtrees with mixed full and empty leaves in step (1) of the algorithm from Section 4.1. To do so, Hsu and McConnell extend the categorization of the leaves as either full or empty to the inner nodes of the tree as follows; see also Figure 8.1 (a). An inner node is *full*, if all but one of its adjacent subtrees, that is the separate trees created by removing the node, have only full leaves. An inner node is *partial*, if it has at

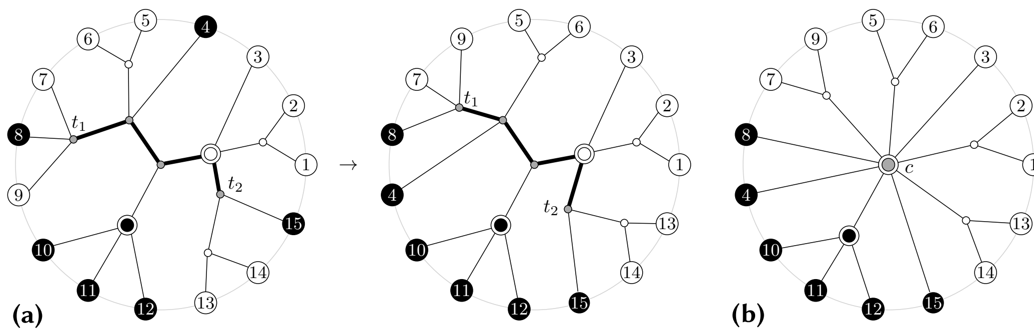


Figure 8.1: (a) Two equivalent PC-Trees with their nodes colored according to the restriction $\{4, 8, 10, 11, 12, 15\}$. C-nodes are represented by big double circles and the P-nodes are represented by small circles. The thick edges represent the terminal path with terminal nodes t_1 and t_2. The white nodes represent empty nodes, the black nodes represent full nodes and the gray nodes represent partial nodes. As the restriction is possible, all full leaves of the tree on the left can be made consecutive, as shown on the right. Furthermore all nodes that must be modified lie on a path. **(b)** Updated PC-tree with new central C-node c.

least one full neighbor and two or more non-full neighbors. Otherwise, that is without a full neighbor, an inner node is *empty*. Then, an edge is terminal if and only if it lies on a path between two partial nodes [HM03; HM04]. Note that the terminal path may contain empty nodes, but cannot contain full nodes, because no full node can be on a path between partial nodes.

Hsu and McConnell incrementally compute this labeling as follows. All inner nodes are initially considered empty, while the leaves are labelled full or empty according to their containment in R. An inner node turns partial once it has at least one full neighbor. A partial node turns full once all but one of its neighbors are full.[13] Assigning the labels and subsequently finding the terminal edges can be done by two bottom-up traversals of the tree, first choosing an arbitrary node of the tree as root. In the following section, we discuss in greater detail how these steps can be implemented.

We summarize the update steps in the following, more fine-granular description of Hsu and McConnell's algorithm for updating the PC-tree [HM04, Algorithm 32.2]. We split step (1) from the high-level description of Section 4.1 to separately represent the two traversals needed to find the terminal path. Note that step (1) was originally merged with step (2) to form the first step in Hsu and McConnell's description. We adopted step (4) being split in two from this original description.

13 Note that all neighbors can only be full if the restriction makes all leaves consecutive, a case which is trivially excluded.

Algorithm for Applying Restrictions

To add a new restriction R to a PC-tree T and compute the updated PC-tree $T + R$:

(1a) Label all partial and full nodes by searching the tree bottom-up from all full leaves.

(1b) Find the terminal path by walking the tree upwards from all partial nodes in parallel.

(2) Perform flips of C-nodes and modify the cyclic order of edges incident to P-nodes so that all full nodes lie on one side of the path.

(3) Split each node on the path into two nodes, one incident to all edges to full nodes and one incident to all edges to empty nodes.

(4a) Delete the edges of the path and replace them with a new C-node c, adjacent to all split nodes, whose cyclic order preserves the order of the nodes on this path.

(4b) Contract all edges from c to adjacent C-nodes, and contract any node that has only two neighbors.

8.2 Our Implementations

The main challenge posed to the data structure for representing the PC-tree is that, in step (4b), it needs to be able to merge arbitrarily large C-nodes in constant time for the overall algorithm to run in linear time. This means that, whenever C-nodes are merged, updating the pointer to a persistent C-node object on every incident edge would be too expensive. Hsu and McConnell (see [HM04, Definition 32.1]) solve this problem by using C-nodes that, instead of having a permanent node object, are only represented by the doubly-linked list of their incident half-edges, which we call *arcs*. This complicates various details of the implementation, like finding the parent pointer of a C-node, which are only superficially covered in the initial work of Hsu and McConnell [HM03]. These issues are in part remedied by the so called *block-spanning pointers* introduced in the later published book chapter [HM04], which are related to the pointer borrowing strategy introduced by Booth and Lueker [BL76]. These block-spanning pointers link the first and last arc of a consecutive block of full arcs (i.e. the arcs to full neighbors) around a C-node and can be accompanied by temporary C-node objects, see the blue dashed arcs in Figures 8.2, 8.6 (c) and 8.6 (d) for an example. Whenever a neighbor of a C-node

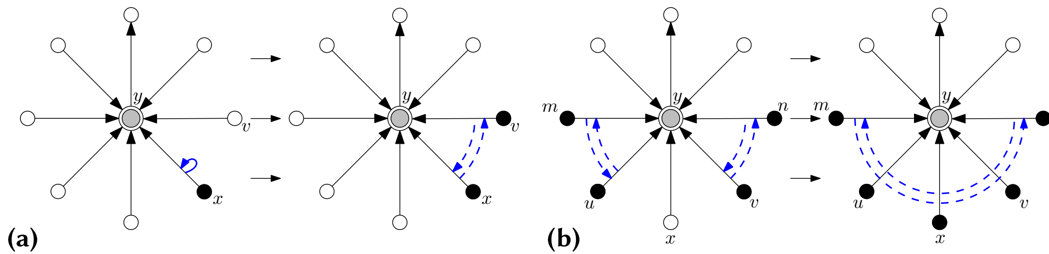


Figure 8.2: (a) A newly created block x of size one growing by one as the neighboring arc v becomes full and is appended. **(b)** Two blocks that share a common neighbor x being merged once x becomes full. The block-spanning pointers are shown as blue, dashed half-arcs.

becomes full, either a new block is created for the corresponding arc of the C-node (Figure 8.2 (a) left), an adjacent block grows by one arc (Figure 8.2 (a) right), or the two blocks that now became adjacent are merged (Figure 8.2 (b)).

Using this data structure, Hsu and McConnell show that the addition of a single new restriction R takes $O(p + |R|)$ time, where p is the length of the terminal path, and that applying restrictions $R_1, \ldots, R_k$ takes $\Theta(|L| + \sum_{i=1}^{k} |R_i|)$ time [HM03; HM04]. Especially for steps (1a) and (1b), they only sketch the details of the implementation, making it hard to directly put it into practice. In the following subsections, we fill in the necessary details for these steps and also refine their running time analysis, showing that step (1a) can be done in $O(|R|)$ time and step (1b) can be done in $O(p)$ time. Using the original procedures by Hsu and McConnell, steps (2) and (3) can be done in $O(|R|)$ time and steps (4a) and (4b) can be done in $O(p)$ time.

For our first implementation, which we call HsuPC, we directly implemented these steps in C++, using the data structure without permanent C-node objects as described by Hsu and McConnell; see Figure 8.3 (a). During the evaluation, we realized that traversals of the tree are expensive. This is plausible, as they involve a lot of pointer-dereferencing to memory segments that are not necessarily close-by, and therefore lead to cache misses. To avoid additional traversals for clean-up purposes, we store information that is valid only during the update procedure with a timestamp. Furthermore, we found that keeping separate objects for arcs and nodes and the steps needed to work around the missing C-node objects pose a non-negligible overhead.

To remove this overhead, we created a second version of our implementation, which we call UFPC, using a Union-Find tree [TL84] for representing C-node objects: Every C-node is represented by an entry in the Union-Find tree and every

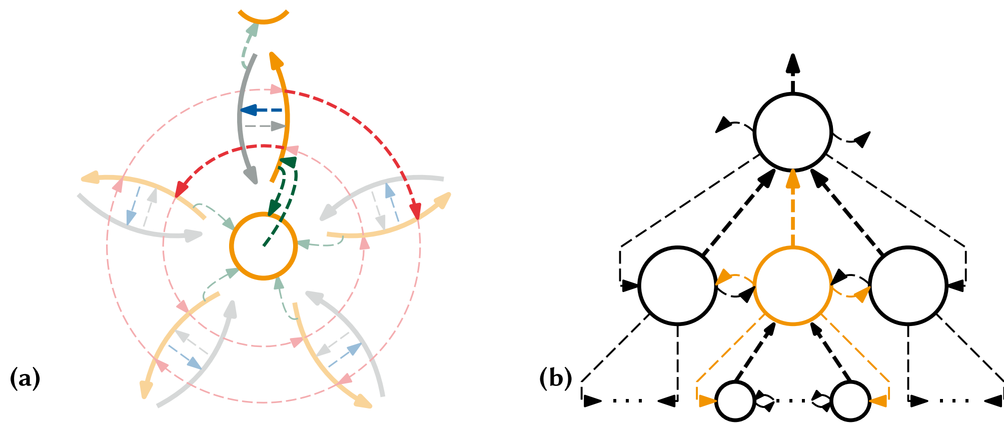


Figure 8.3: Visualization of the data structures used by HsuPC (left) and UFPC (right) for representing the PC-tree. In-memory objects are shown with solid lines while pointers are represented by dashed arrows. **(a) HsuPC:** The orange circle represents a P-node object, the outwards-facing orange arrows are the arc objects representing its outgoing incident half-edges. The arc objects for the reverse direction are shown in gray and each arc has a pointer to its twin (the blue and grey dashed arrows). Furthermore, each arc knows its predecessor and successor around its node (the red dashed arrows). Each arc leaving a P-node has a reference to the P-node object and each non-root P-node object has a reference to the arc leading to its parent (the green arrows). Note that for C-nodes, the green arrows are null-pointers as no node object exists. The parent P-node of the current P-node is partially shown at the top. **(b) UFPC:** A selected P-node object is highlighted in orange. It has pointers to its parent (upwards), siblings (left and right) and first and last child (downwards left and right). Note that if the parent is a C-node, the parent pointer instead points into the Union-Find structure.

incident child edge stores a reference to this entry. Whenever two C-nodes are merged, we apply union to both entries and only keep the object of the entry that survives. This leads to every lookup of a parent C-node object taking amortized $O(\alpha(|L|))$ time, where α is the inverse Ackermann function. Although this makes the overall running time super-linear, the experimental evaluation in Section 8.3 shows that this actually improves the performance in practice. As a second change, the UFPC no longer requires separate arc and node objects, allowing us to use a more lightweight doubly-linked tree consisting entirely of nodes that store pointers to their parent node, left and right sibling node, and first and last child node; see Figure 8.3 (b). Edges are represented implicitly by the child node whose parent is the other end of the edge. Note that of the five stored pointers, a lookup in the Union-Find data structure is only needed for resolving the parent of a node.

We use the Union-Find data structure from the OGDF [Chi+14] and plan to

merge our UFPC implementation into the OGDF. Furthermore, both our C++ implementations should also be usable stand-alone with a custom Union-Find implementation. The source code for both implementations, our evaluation harness and the code for generating all test data are available on GitHub (see Table 8.1).

In the following, we describe further details in which our implementations differ from the description given by Hsu and McConnell and explain the corrections needed for a working implementation. Section 8.2.1 describes how the labeling procedure (step (1a)) can be properly implemented. Note that the technical complications involving arcs and block-spanning pointers only concern the missing C-node objects of HsuPC. UFPC uses direct references to adjacent C-nodes instead of arcs and does not need to maintain block-spanning pointers. Thus, those parts can be greatly simplified for our second implementation, but we still give the full details in our description using the perspective of the HsuPC data structure. The same holds for Section 8.2.2, where we give a corrected algorithm for enumerating the terminal path (step (1b)). Section 8.2.3 then describes the generic steps needed to detect impossible restrictions. Lastly, Section 8.2.4 explains the differing update procedure of UFPC (steps (4a) and (4b)), while the update procedure of HsuPC follows Hsu and McConnell's original description.

8.2.1 Efficiently Labeling and Finding Partial Nodes

In our description of the labeling step, we follow the general procedure of Hsu and McConnell [HM04], which uses a bottom-up traversal of the tree, starting at the full leaves. Algorithm 3 gives the pseudo-code for this procedure. Initially, all inner nodes are considered empty. Recall that arcs only have a reference to their target node if it is a P-node. We thus need to be careful when following edges as there might not exist an object for the (C-)node at the other end of the edge. To implement the traversal, we keep a queue of unprocessed arcs pointing from full nodes to non-full neighbors. Furthermore, each P-node object stores a list of incoming arcs from full neighbors. At the start of the traversal, the queue is initialized with the incident arcs of all full leaves. If the arc at the front of the queue has a reference to its target node object (via its twin arc; see Figure 8.3 (a)), this has to be a P-node (as only these are represented by actual objects) and we can simply append the incoming arc to the P-node's list of full children. Recall that each full node has exactly one non-full neighbor. Thus, if this list reaches a size one smaller than the P-node's degree, we mark it as full and enqueue the pointer to the single non-full neighbor of the P-node. Usually, this non-full neighbor is the parent of the P-node, so we directly enqueue the parent arc if it is not null and the parent is not yet full. The other case, when the now-full P-node is the root or when a parent

Algorithm 3: Label full and partial nodes given a set of consecutive leaves R.

```
LabelNodes(R):
1  Q ← {}; // A queue of unprocessed arcs.
2  for l ∈ R do
3      label leaf l as full;
4      add the single incident arc of l to queue Q;
5  while Q is not empty do
6      remove next arc a from Q;
7      mark the twin arc of a as leading to a full node;
8      if a has a reference to its target node u then // u must be a P-node
9          add a to u's list of full neighbors F_u and mark u as partial;
10         if |F_u| = deg(u) − 1 then // P-node became full
11             mark u as full;
12             if u has a parent arc that is not full then
13                 add u's parent arc to Q; // optimization for the common case
14             else
15                 search through all arcs incident to u to find the single
                     non-full arc f;
16                 add f to Q;
17     else // manage blocks at C-node
18         create/append/merge the full blocks adjacent to a, yielding a full
             block b;
19         if both ends of b are adjacent to a single non-full arc f then
20             mark b (as proxy for the missing C-node object) as full;
21             add f to Q; // C-node became full
22         else
23             mark b (as proxy for the missing C-node object) as partial;
```

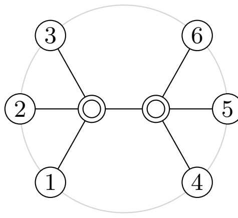


Figure 8.4: The PC-tree with six leaves representing the restrictions $\{\{1,2\},\{2,3\},\{4,5\},\{5,6\}\}$, which consists of two adjacent C-nodes. Up to reversal and cyclic shifting, it only has the two admissible orders $[1,2,3,4,5,6]$ and $[1,2,3,6,5,4]$.

has become full before its child P-node, is missing in the description by Hsu and McConnell. Here, we need to search all incident arcs for the single arc pointing to a non-full node, and queue this arc instead. As the total number of searched arcs is bounded by the number of full leaves, this does not affect the overall running time.

If the arc at the front of the queue does not have a reference to a (P-)node, we need to maintain (i.e., create/append/merge) the block-spanning pointers around the respective C-node x. We will also use the full blocks managing these block-spanning pointers as proxy objects for partial C-nodes. The merging of full blocks is illustrated in Figure 8.2, see the book chapter by Hsu and McConnell [HM04] for more details. If the new endpoints of the C-node's full block are now adjacent to the same arc pointing to a non-full neighbor, the C-node is full and we queue the arc to this neighbor z. Note that similar to the case of P-nodes, this node is most often but not necessarily (e.g., if the root became full) the parent arc. Still, there is no explicit search required, as we already know the arc to the only non-full neighbor. Once the queue runs empty, the labeling is complete. The partial nodes are now represented by the non-full P-nodes that have full neighbors (but at least two non-full neighbors) and the full blocks around non-full C-nodes (i.e., where both endpoints are not adjacent to the same arc).

Finally, let us make a further minor correction regarding the original description. In their definition of the data structure Hsu and McConnell note that "no two C nodes are adjacent, so each of these edges [incident to a C-node] has one end that identifies a neighbor of the C node, and another end that indicates that the end is incident to a C node, without identifying the C node" [HM04, page 32-10]. See Figure 8.4 for a simple counterexample where two C-nodes are indeed adjacent. Hsu and McConnell use this property within their planarity test and after they test whether the endpoints of a full block of a C-node x are adjacent to the same arc leading to a neighbor z: "if x passes this test, it is full, and the full-neighbor counter of z is incremented" [HM04, page 32-11]. According to their argumentation, z has to be a P-node as no two C nodes are adjacent, which is incorrect. Still, the important information is not the type of the nodes, but that the neighbor x of the

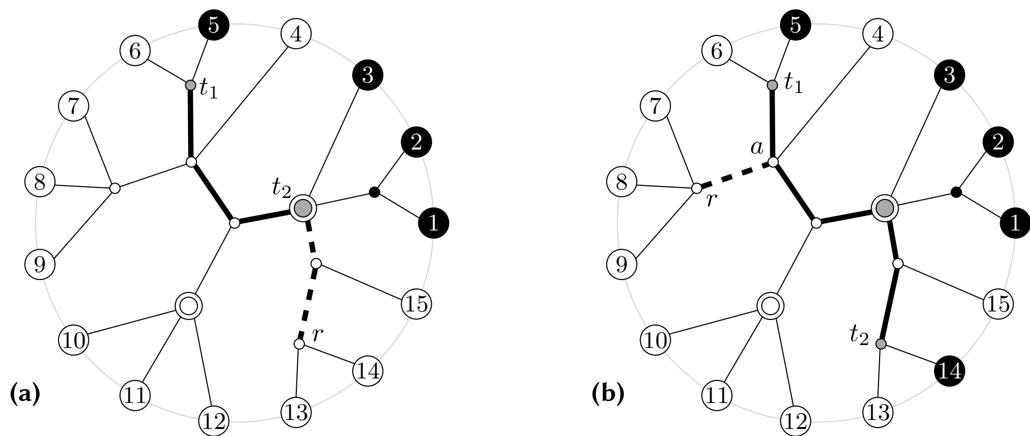


Figure 8.5: Two examples of a PC-tree with root r. **(a)** An I-shaped terminal path where t_2 is both apex and terminal node. **(b)** An A-shaped terminal path where two ascending paths join in the apex a.

non-full node z became full and we thus need to queue the arc from x to z. Thus, our queue-based approach also correctly handles the case where a chain of multiple adjacent C-nodes becomes full in a cascading fashion.

8.2.2 Efficiently Finding the Terminal Path

Hsu and McConnell show that an edge is terminal if and only if it lies on a path in the tree between two partial nodes. This allows them to conduct parallel searches, starting at every partial node and extending ascending paths through their ancestors at the same rate [HM04]. Whenever an already processed node is encountered, expansion of the current path is stopped and the path is instead merged into the path of the already processed node. Once all paths have met, the search can be terminated. There are two possible structures for the finished terminal path, assuming the restriction is possible. Let the *apex* be the highest node on the terminal path, i.e., the lowest node that is an ancestor of all other nodes on the terminal path. The two cases can now be differentiated based on the position of the apex, which in turn depends on the position of the root node:

I-Shaped: If the apex lies on one of the ends of the terminal path and is therefore a terminal node at the same time, the terminal path extends from the other endpoint of the path upwards to the apex, as shown in Figure 8.5 (a). In this

case, every node on the terminal path has exactly one child on the terminal path, except for the lower terminal node t_1. This also covers the special case where there is only a single terminal node $t_1 = t_2$.

A-Shaped: If the apex does not lie on one of the ends of the terminal path, two ascending paths join in the apex, as shown in Figure 8.5 (b). In this case, the apex a has two children on the terminal path, the terminal nodes t_1 and t_2 have none, and all other nodes on the terminal path have exactly one child that is also on the terminal path.

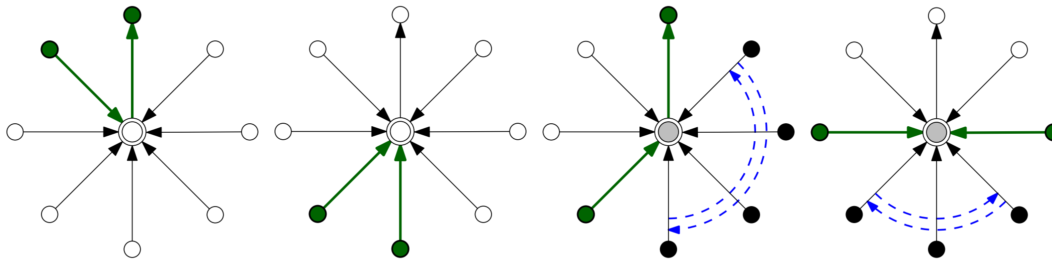
The apex can be found using a bottom-up traversal, this time starting from all partial nodes found during the labeling step. As before, traversing a P-node can be done easily, but again as C-nodes have no object registered with their incident arcs, finding their parent arc can be difficult. To do so, Hsu and McConnell distinguish two cases depending on whether they arrived at the C-node via an arc from a partial or empty node, or via an arc from a full node. We note that the parallel searches can never actually ascend to another node coming from a full node, as a full node cannot be part of the terminal path. Thus, we focus exclusively on the case where we arrived at a C-node from a partial or empty neighbor. Here, Hsu and McConnell "look at the two neighbors of the child edge in the cyclic order, and one of them must be the parent edge." [HM04, page 32-12]. This is only correct in the first case of the four cases shown in Figure 8.6, as the parent arc may lie behind a full block adjacent to the incoming arc or the current node may be the apex of the terminal path.

Algorithm 4 shows our corrected procedure for finding the parent arc for any given arc a. It either returns said arc, detects that we can stop ascending or aborts the algorithm as the restriction is impossible. If the parent arc of an empty C-node is part of an I-shaped terminal path, it has to lie next to the incoming terminal path arc a, otherwise the empty neighbors would be non-consecutive and the restriction thus impossible; see Figure 8.6 (a) and lines 11 to 14 of Algorithm 4. If the parent arc of a partial C-node is part of an I-shaped terminal path, it has to lie on the opposite side of the full block adjacent to the incoming terminal path arc a for the restriction to be possible; see Figure 8.6 (c) and lines 17 and 21. If this arc is not the parent arc, it may still be part of the terminal path if the current node is the apex of an A-shaped terminal path as shown in Figures 8.6 (b) and 8.6 (d), see also line 15 as well as lines 18 and 22. Note that in this case, the second incoming terminal path arc will only be found at a later iteration, as we cannot identify the shape or position of the apex beforehand. If we neither find a parent arc nor a second incoming arc, ascending through the current node would make the current restriction impossible as full

Algorithm 4: Process an arc a on the terminal path to either return the parent of its target node, detect that we can stop ascending and return null, or abort the algorithm as the restriction is impossible.

FindParentArc(a):

```
1  if a has a reference to its target node u then // u must be a P-node
2      if u is full then
           // we already ascended too far
3      else if u has a parent arc then
4          return parent arc of u;
5      else
           // we reached the root and cannot ascend further
6  else // C-node
7      a1, a2 ← the two arcs adjacent to a;
8      b1, b2 ← the full blocks ending at a1, a2 or null;
9      if b1 = null then
10         if b2 = null then // no blocks adjacent
               // we can ascend if parent arc is adjacent to a; see Fig. 8.6 (a)
11             if a1 leads to parent then
12                 return a1;
13             else if a2 leads to parent then
14                 return a2;
15             else
                   // no parent nearby, we cannot ascend further; see Fig. 8.6 (b)
16         else if parent arc is at other side of b2 then
17             return parent arc adjacent to b2 similar to above; // see Fig. 8.6 (c)
18         else
               // no parent nearby, we cannot ascend further; see Fig. 8.6 (d)
19     else if b2 = null then
20         if parent arc is at other side of b1 then
21             return parent arc adjacent to b1; // see Fig. 8.6 (c)
22         else
               // no parent nearby, we cannot ascend further; see Fig. 8.6 (d)
23     else if b1 = b2 then
           // the C-node is full and we ascended too far
24     else // two different full blocks adjacent
25         raise impossible restriction;
26 return null; // we cannot ascend further
```



(a) An *empty* C-node that is part of a possibly *I-shaped* terminal path. **(b)** An *empty* C-node that is *apex* of an *A-shaped* terminal path. **(c)** A *partial* C-node that is part of a possibly *I-shaped* terminal path. **(d)** A *partial* C-node that is *apex* of an *A-shaped* terminal path.

Figure 8.6: Different cases of the terminal path crossing a C-node. Empty, partial, and full nodes are drawn in white, gray, and black, respectively. The thicker, green nodes and edges are part of the terminal path, the blue dashed half-arcs depict the block-spanning pointers. The edges are oriented towards the root node. In case (c), the node can be a final node of the terminal path, i.e. a terminal node. Thus there may be zero, one or two incident terminal edges. In cases (b), (c), and (d), the node can also be the root, i.e. lacking a parent arc.

and empty leaves could then not be separated on different sides of the terminal path. We can thus simply stop ascending at any C-node for which we did not find a parent (line 26). Similarly, we stop ascending if we arrive at a full node (which cannot be part of the terminal path) or the root node (lines 2, 5 and 23).

To now correctly enumerate the terminal path, we again use a queue of unprocessed arcs; see Algorithm 5. We initialize the queue with the parent arcs of all partial nodes found in the labeling step, which is easy to do for partial P-nodes; see lines 15 to 23 of Algorithm 5. For partial C-nodes, we can check for a parent arc that is adjacent to the respective full block, see Figure 8.6 (c) and line 6. If we are unable to find a parent for the current partial node, we store it both as apex and as highest point of a stopped search path (lines 12 and 14).

We process the queue arc by arc, using FindParentArc from Algorithm 4 to find the parent arc of each dequeued arc (lines 24, 25, and 31). Recall that we never process an arc twice by merging the respective search paths once we encounter an already processed arc (lines 26 and 27). If we are unable to find a parent for the current arc, we store it as highest arc of a stopped search path (line 43). We report an impossible restriction if multiple search paths stopped early, as the terminal path would then be disconnected. This is handled by the SetApex method in lines 12, 21, and 45; see Section 8.2.3 for more details on impossible restrictions. We also stop processing if there is only one arc left in the queue and we have not yet

Algorithm 5: Enumerate the terminal path as a list of predecessors of the apex.

```
EnumerateTP():
1  apex ← null; // A partial P-node or block of a partial C-node in case of an
     I-shaped apex, a pair of terminal path arcs pointing to the same node in
     case of an A-shaped apex.
   // The chain of predecessors of the apex forms the terminal path. Property
      partial-predecessor indicates the highest partial node on the terminal path
      below, and that an arc is part of the terminal path.
2  foreach arc a do a.predecessor ← null; a.partial-predecessor ← null;
3  highest ← null; // The node, block or arc at which a search path stopped.
     Knows the highest partial node on said path as its partial-predecessor.
4  Q ← {}; // A queue of unprocessed arcs.
   // Initialize Q with partial nodes
5  foreach block b at a partial C-node do
6      if the parent arc p of the C-node lies directly before or after b then
7          if p.partial-predecessor ≠ null then
8              raise impossible restriction; // see Fig. 8.7 (a) and Sec. 8.2.3
9          p.partial-predecessor ← b;
10         add p to queue Q;
11     else
12         SetApex(b); // I- or A-shaped
13         b.partial-predecessor ← b;
14         highest ← b;
15 foreach partial P-node u do
16     p ← parent of u;
17     if p ≠ null then
18         p.partial-predecessor ← u;
19         add p to queue Q;
20     else
21         SetApex(u); // I- or A-shaped
22         u.partial-predecessor ← u;
23         highest ← u;
   // continued on next page...
```

```
   // Main Routine: Process Queue
24 while Q is not empty do
25     remove next arc a from Q;
26     if a was already visited then
27         continue;
28     if Q reached length 0 and highest = null then
           // search paths already converged, we are extending above apex
29         highest ← a;
30         break;
31     p ← FindParentArc(a);
32     if p ≠ null then
33         if p.predecessor = null then
34             if p.partial-predecessor ≠ null then
35                 raise impossible restriction; // see Fig. 8.7 (b) and Sec. 8.2.3
36             p.predecessor ← a;
37             p.partial-predecessor ← a.partial-predecessor;
38             add p to Q;
39         else
40             SetApex((a, p.predecessor)); // A-shaped
41     else
42         if highest = null then
43             highest ← a;
44         else
45             SetApex((a, highest)); // A-shaped
46 if apex = null then
       // I-shaped, the apex is the highest partial node
47     SetApex(highest.partial-predecessor);
```

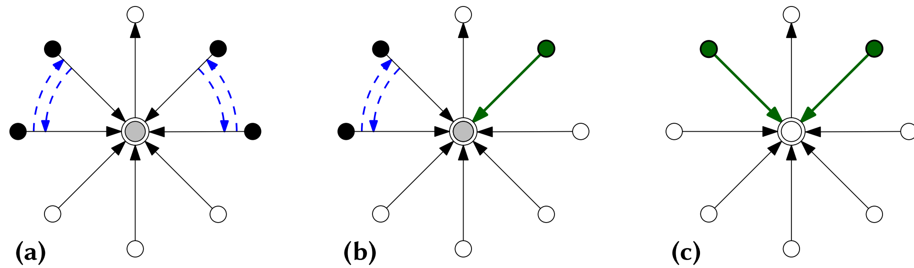


Figure 8.7: Examples of impossible restrictions where the parent arc would be used by two different paths. **(a)** A partial C-node with two full blocks next to its parent edge. **(b)** A partial C-node with one full block and one terminal path edge next to its parent edge. **(c)** An empty C-node with two terminal path edges next to its parent edge.

stopped a search path at a highest point (for which we could not find a parent arc); see lines 28 to 30. In this case, all parts of the parallel search have already converged into a single ascending path and we are extending the terminal path above the actual apex.

The apex will be the node that has two incident terminal edges in case of an A-shaped terminal path, or the highest partial node in case of an I-shaped terminal path. The first case can be identified by checking whether a node has two predecessors on the terminal path; see lines 33 and 40. The backtracking needed in the second case can be done in constant time (lines 46 and 47) by storing the highest partial predecessor for each processed arc (lines 9, 13, 18, 22, and 37).

Observe that the number of arcs on the terminal path is proportional to the length p of the terminal path. Furthermore, we only check a constant number of neighbors of each arc and any highest arc requiring backtracking is at most p nodes above the actual apex. Thus, the overall running time of our search for the terminal path is in $O(p)$ if the restriction is possible. Note that this slightly refines the analysis of Hsu and McConnell [HM04], who sometimes scan the full children of a node and thus have a running time in $O(p + |R|)$.

8.2.3 Efficiently Detecting Impossible Restrictions

Recall that a restriction is possible if and only if (1) all terminal edges form a path and (2) all nodes on the terminal path can be flipped or rearranged such that their empty and full children are consecutive while separated by the terminal edges. The only way to violate the first property is when a node has more than two incident terminal edges. P-nodes can detect directly when this case occurs, while a C-node with more than two incident terminal edges leads to multiple stopped search paths,

Algorithm 6: Set a P-node, full block around a C-node, or a pair of arcs o to be the apex or report an invalid restriction.

```
SetApex(o):
1  if o is a pair of arcs (a, b) then // validate an A-shaped apex
2      if a or b point to a P-node then
3          if a and b do not point to the same P-node then
4              raise impossible restriction;
5      else // C-node
6          if a and b do not lie next to each other // see Figure 8.6 (b)
7             or a and b do not lie next to the same full block // see Figure 8.6 (d)
8           then
9              raise impossible restriction; // see Figure 8.7 (c)
10 if apex is a P-node then
11     if o is the pair of arcs pointing to apex then
12         apex ← o;
13     else
14         raise impossible restriction;
15 else if apex is a block at a C-node then
16     if o is the pair of arcs that lie before and after apex then
17         apex ← o;
18     else
19         raise impossible restriction;
20 else if apex is a pair of arcs then
21     raise impossible restriction;
22 else // apex = null
23     apex ← o;
```

as the parent edge of a C-node is only found when it is next to the incoming terminal path edge. This is detected by our algorithm when calling SetApex multiple times with different highest arcs.

As P-nodes allow arbitrary arrangements of their children, only C-nodes can violate the second property by either having multiple distinct full blocks, by having a full block that is not adjacent to all incident terminal edges, or by having no full block and two or more non-adjacent terminal edges. All these cases lead to a disconnected terminal path and thus multiple stopped search paths. In our pseudo-code there is one exception to this: Two full blocks or terminal path edges before and after the parent edge of a C-node could both independently use the parent edge for ascending; see Figure 8.7. We detect this situation with two full blocks, one full block and one terminal path edge, and two terminal paths in lines 8 and 35 of Algorithm 5 and line 9 of Algorithm 6, respectively. Here, we use the node property partial-predecessor, which is also used to efficiently find the apex when we extended an I-shaped terminal path too far, to detect that an edge has already been used for another path.

Finally, note that our pseudo-code may identify the same node as apex twice if it is a partial node for which we could not find a parent arc and it also is apex of an A-shaped terminal path, i.e., has two predecessors on the terminal path; see lines 11 and 16 of Algorithm 6. Otherwise, we report an impossible restriction once we encounter a second apex (or highest arc, i.e. stopped search path) as shown in Algorithm 6.

8.2.4 Deletion and Contraction

Deleting and inserting new edges is simple when using the arc-based tree representation described by Hsu and McConnell. When using a doubly-linked tree structure similar like the one used by UFPC, no explicit edge objects exists and they are instead encoded by the child-parent relationship of the nodes. This means that for the deletions and contractions in steps (4a) and (4b) of the overall update procedure, the child-parent relationship needs to be set immediately and correctly for every change and cannot easily be updated later, as done by Hsu and McConnell. Thus, UFPC uses a different approach for these two steps, which is conceptually closer to our initial description of step (4) in Section 4.1. First, when creating the central node, we need to make sure that we directly assign it its neighbors. This is trivial if the apex is a C-node and thus can simply be reused as is. Otherwise, we create a new C-node and add up to four neighbors: the apex' first child on the terminal path, a newly created P-node that was reassigned as parent of all full children of the apex, the second child on the terminal path, and finally the apex with all its empty

children remaining. Here, neighbors that do not exist are left out, e.g., when the apex has no full or empty children or the terminal path is I-shaped. Furthermore, the root of the tree is either among the full or the empty children and thus the node that is still connected to the root needs to be installed as parent of the new central node. Second, we iteratively contract a child of the central node that is part of the terminal path into the central node. C-nodes can again be simply merged, while a P-node x needs to be split into a full and an empty node. P-node x is then replaced by the full node and the empty node with the other terminal path neighbor of x in between, if the latter exists.

8.3 Evaluation

In this section, we experimentally evaluate our PC-tree implementations by comparing the running time for applying a restriction with that of various PQ- and PC-tree implementations that are publicly available. In the following we describe our methods for generating test cases, our experimental setup and report our results.

8.3.1 Test Data Generation

To generate PQ-trees and restrictions on them, we use the planarity test by Booth and Lueker [BL76], one of the initial applications of PQ-trees. This test incrementally processes vertices one by one according to an st-ordering. Running the planarity test on a graph with n vertices applies $n-1$ restrictions to PQ-trees of various sizes. Since not all implementations provide the additional modification operations necessary to implement the planarity test, we rather export, for each step of the planarity test, the current PQ-tree and the restriction that is applied to it as one instance of our test set. We note that the use of st-orderings ensures that the instances do not require the ability of PC-trees to represent cyclic instead of linear orders, which makes them good test cases for comparing PC-trees and PQ-trees.

In this way, we create one test set SER-POS consisting of only PQ-trees with possible restrictions by exporting the instances from running the planarity test on a randomly generated biconnected planar graph for each vertex count n from 1000 to 20 000 in steps of 1000 and each edge count $m \in \{2n, 3n-6\}$.[14] Altogether,

14 Note that simple graphs with more than $3n-6$ edges are always non-planar, while connected graphs with only n edges are always planar. Furthermore maximal planar graphs, i.e., those with exactly $3n-6$ edges, have an up to mirroring unique planar embedding. Thus, we chose $2n$ edges as a natural middle ground between being trivially planar and having no embedding choices.

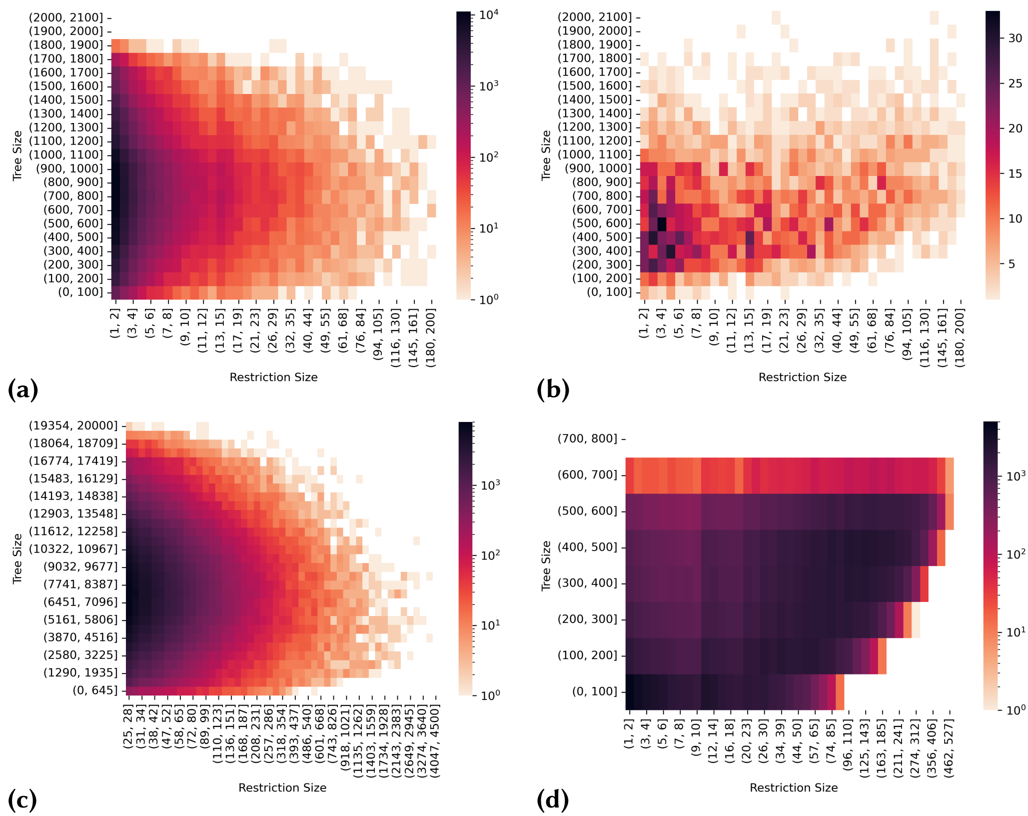


Figure 8.8: Distribution of tree and restriction size for the data sets **(a)** SER-POS, **(b)** SER-IMP, **(c)** DIR-PLAN, and **(d)** MAT-POS. The colors indicate how many instances each cell contains. Please note the different axis and color scales.

this test set contains 199 831 instances, whose distribution with regards to tree size, restriction size, and terminal path length is shown in Figures 8.8 (a) and 8.9 (a).

To guard against overly permissive implementations, we also create a small test set SER-IMP of impossible restrictions. It is generated in the same way, by adding randomly chosen edges to the graphs from above until they become non-planar. In this case the planarity test fails with an impossible restriction at some point; we include these 3800 impossible restrictions in the set; see Figures 8.8 (b) and 8.9 (b).

As most of the available implementations have no simple means to store and load a PQ-/PC-tree, we serialize each test instance as a set of restrictions that create the tree, together with the additional new restriction. When running a test case, we first apply all the restrictions to reobtain the tree, and then measure the time to apply the new restriction from the test case. The prefix SER- in the name of both sets emphasizes this serialization.

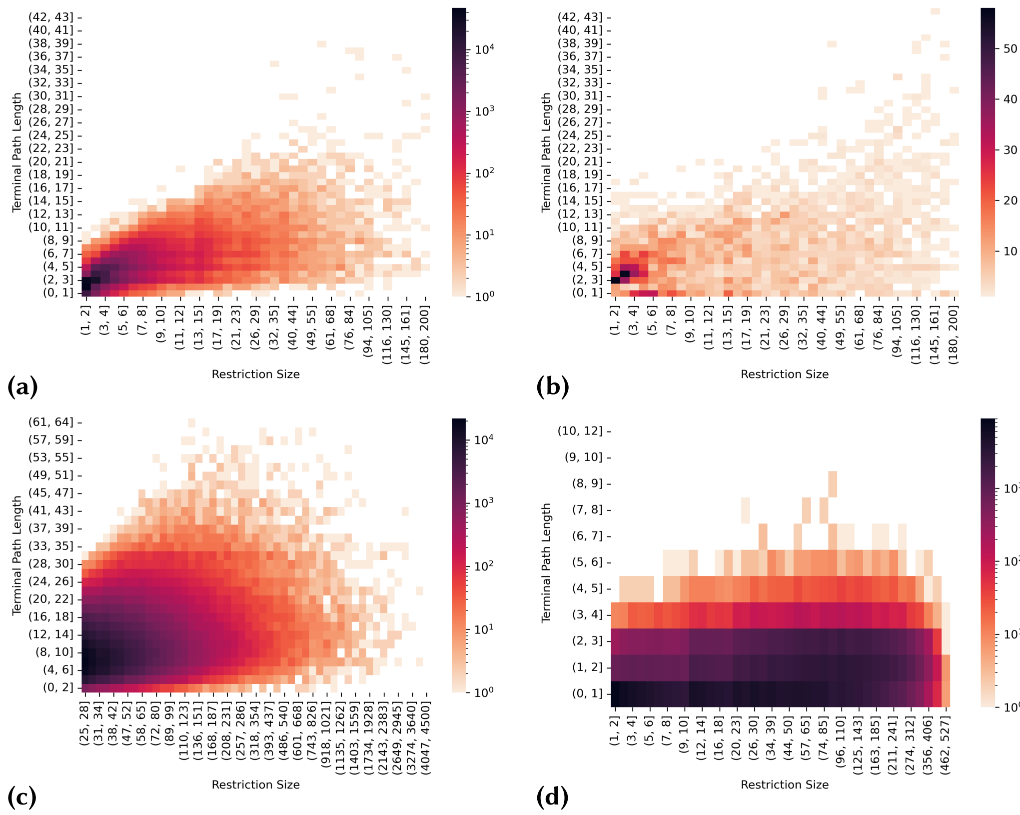


Figure 8.9: Distribution of terminal path length and restriction size for the data sets **(a)** SER-POS, **(b)** SER-IMP, **(c)** DIR-PLAN, and **(d)** MAT-POS.

To be able to conduct a more detailed comparison of the most promising implementations, we also generate a third test set with much larger instances. As deserializing a PC- or PQ-tree is very time-consuming, we directly use the respective implementations in the planarity test by Booth and Lueker [BL76], thus calling the set DIR-PLAN. We generated 10 random planar graphs with n vertices and m edges for each n ranging from 100 000 to 1 000 000 in steps of 100 000 and each $m \in \{2n, 3n - 6\}$, yielding 200 graphs in total. The planarity test then yields one possible restriction per node. As we only want to test large restrictions, we filter out restrictions with less than 25 full leaves, resulting in DIR-PLAN containing 564 300 instances; see Figures 8.8 (c) and 8.9 (c).

In order to evaluate additional test cases that stem from a different application, we also create a fourth test set MAT-POS containing instances of the consecutive ones problem. To this end, we generate 1000 matrices in $\{0, 1\}^{m \times n}$ with m, n chosen

uniformly at random from the interval $[10, 500]$. We pick a random range of cells as the consecutive block of ones in each row and then shuffle the columns of the matrix, which yields a yes-instance of the consecutive ones problem. We also add a zero column to each matrix to ensure that the PC- and PQ-trees represent the same admissible orders. In total, the set MAT-POS contains 249 080 restrictions whose distribution is shown in Figures 8.8 (d) and 8.9 (d).

8.3.2 Experimental Setup

Table 8.1 gives an overview of all implementations we are aware of, although not all implementations could be considered for the evaluation. The three existing implementations of PC-trees we found are incomplete and unusable (Luk&Zhou) or tightly intertwined with a planarity test in such a way that we were not able to extract a generic implementation of PC-trees (Hsu, Noma). We further exclude two PQ-tree implementations as they either crash or produce incorrect results on almost all inputs (GTea) or have an excessively poor running time (TryAlgo). Among the remaining PQ-tree implementations only two correctly handle all our test cases (OGDF, SageMath). Several other implementations have smaller correctness issues: After applying a fix to prevent segmentation faults in a large number of cases for BiVoC, the remaining implementations crash (BiVoC, GraphSet, Zanetti, Gregable) and/or produce incorrect results (Reisle, JGraphEd, Zanetti) on a small fraction of our tests; compare the last column of Table 8.1. We nevertheless include them in our evaluation. We changed the data structure responsible for mapping the input to the leaves of the tree for BiVoC and Gregable from `std::map` to `std::vector` to make them competitive. Moreover, BiVoC, Gregable and GraphSet use a rather expensive cleanup step that has to be executed after each update operation. As this could probably largely be avoided by the use of timestamps, we do not include the cleanup time in their reported running times. For SageMath the initial implementation turned out to be quadratic, which we improved to linear by removing an unnecessary recursion. As Zanetti turned out to be a close competitor to our implementation in terms of running time, we converted the original Java implementation to C++ to allow a fair comparison (CppZanetti). This decreased the running time by one third while still producing the exact same results. All other non-C++ implementations were much slower or had other issues, making a direct comparison of their running times within the same language environment as our implementations unnecessary. Further details on the implementations are given at the end of this section.

Table 8.1: Implementations considered for the evaluation. Implementations that are entirely unusable as they are incomplete or crash/produce incorrect results on almost all inputs (marked with –) and those where no generic PC-/PQ-tree implementation could be extracted (marked with n.a.) could not be evaluated. Correct implementations are marked with ✓and implementations that are functional, but do not always produce correct results are marked with ✗. These two categories are included in our experimental evaluation. The last column shows the number of errors for the 203 630 restrictions in the sets SER-POS and SER-IMP and the 1000 matrices in the set MAT-POS.

Name	Type	Context	Language	Correct	Errors	URL
HsuPC	PC-Tree	our impl., based on [HM04]	C++	✓	0	https://github.com/N-Coder/pc-tree/tree/HsuPCSubmodule
UFPC	PC-Tree	our impl. using Union-Find	C++	✓	0	https://github.com/N-Coder/pc-tree
Luk&Zhou	PC-Tree	student course project	C++	–	–	https://github.com/kwmichaelluk/pc-tree
Hsu [Hsu03]	PC-Tree	planarity test prototype	C++	n.a.	–	http://qa.iis.sinica.edu.tw/graphtheory
Noma [Boy+04b]	PC-Tree	planarity test evaluation	C++	n.a.	–	https://www.ime.usp.br/~noma/sh
OGDF [Lei97]	PQ-Tree	planarity testing	C++	✓	0	https://ogdf.github.io
Gregable	PQ-Tree	biclustering	C++	✗	1	https://gregable.com/2008/11/pq-tree-algorithm.html
BiVoC [GMM06]	PQ-Tree	automatic layout of biclusters	C++	✗	73	https://bioinformatics.cs.vt.edu/~murali/papers/BiVoC
Reisle	PQ-Tree	student project	C++	✗	252	https://github.com/creisle/pq-trees

Table 8.1: Implementations considered for the evaluation (continued).

Name	Type	Context	Language	Correct	Errors	URL
GraphSet [EFK09]	PQ-Tree	visual graph editor	C++	✗	1551	http://graphset.cs.arizona.edu
Zanetti [Zan12]	PQR-Tree*	extension of PQ-Trees	Java	✗	728	https://github.com/jppzanetti/PQRTree
CppZanetti	PQR-Tree*	our C++ conversion of Zanetti	C++	✗	728	https://github.com/N-Coder/pc-tree#installation
JGraphEd [Har04]	PQ-Tree	visual graph editor	Java	✗	11	https://www3.cs.stonybrook.edu/~algorith/implement/jgraphed/implement.shtml
GTea [CH17]	PQ-Tree	visual graph theory tool	Java	–	–	https://github.com/rostam/GTea
TryAlgo	PQ-Tree	consecutive-ones testing	Python	–	–	https://tryalgo.org/en/datastructures/2017/12/15/pq-trees
SageMath	PQ-Tree	interval graph detection	Python	✓	0	https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/pq_trees.html

* PQR-Trees are a variant of PQ-Trees that can also represent impossible restrictions, replacing any node that would make a restriction impossible by an R-node (again allowing arbitrary permutation). To make the implementations comparable, we abort early whenever an impossible restriction is detected and an R-node would be generated.

Environment

Each experiment was run on a single core of a Intel Xeon E5-2690v2 CPU (3.00 GHz, 10 Cores, 25 MB Cache) with 64 GiB of RAM, running Linux Kernel version 5.10. Implementations in C++ were compiled with GCC 10.2.1 and optimization `-O3 -march=native -mtune=native`. Java implementations were executed on OpenJDK 64-Bit Server VM 11.0.14 and Python implementations were run with CPython 3.9.2. For the Java implementations we ran each experiment several times, only measuring the last one to remove startup-effects and to facilitate optimization by the JIT compiler. We used OGDF version 2020.02 (Catalpa) to generate the graphs from which we derive our test data. We did not analyze the memory consumption of the implementations, as in theory the linear running time also bounds the memory. Furthermore, the size of the used data structures only differs by a small constant factor. In practice, the use of various different libraries also makes it hard to compare the actual amount of memory used.

Details About Evaluated Implementations

BiVoC, Gregable In the implementations of *BiVoC* and *Gregable*, we improved the mapping from the input to the tree's leaves by replacing `std::map` with `std::vector`, as suggested in the code's comments. As a result, this mapping now takes constant time. The `Bubble` method of BiVoC caused segmentation faults due to undefined behavior, because a set iterator is dereferenced and incremented after its corresponding element has been removed. We resolved this issue for our evaluation. Still, the method `qNextChild` of BiVoC sometimes caused program hangs due to undefined behavior, when the past-the-end iterator of an empty set is incremented. In the Gregable repository, the author notes that the code "is known to be buggy on some rare inputs. A believed to be correct, but harder to use version of this code can be found as a library within BiVoC". In our tests, Gregable produced a segmentation fault in the reduce-step on one input, while BiVoC failed for 73 instances. Figure 8.10 shows an example where Gregable's implementation produces an invalid PQ-tree.

GraphSet In the implementation of *GraphSet*, we removed the entanglement with Microsoft Foundation Classes by replacing its data structures with their corresponding variants from the standard library. We were unable to get GraphSet's `Bubble` method to work for our tests. Instead, we used the approach from their quadratic-time variant of Booth and Lueker's planarity test, where they traverse the entire tree before each reduction in order to find

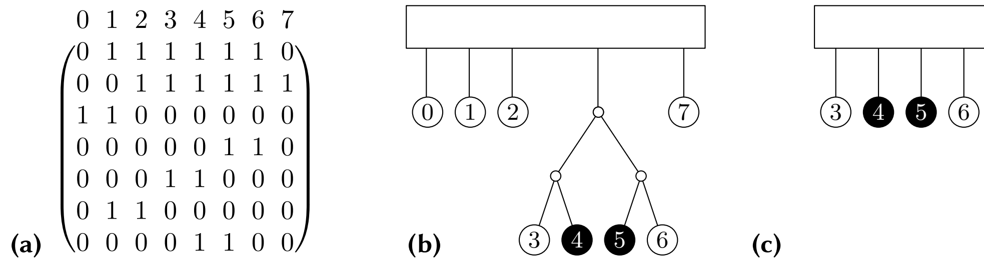


Figure 8.10: (a) A matrix with the consecutive ones property. **(b)** The state of Gregable's PQ-tree data structure before applying the last restriction of the matrix. Q-nodes are depicted as rectangles, P-nodes as small circles. **(c)** The state of the data structure after applying the last restriction {4,5} to the former tree. The new Q-node created in Template P6 is erroneously chosen as the new root node of the tree. Therefore, the tree loses all other leaves.

and prepare the pertinent subtree. Still, GraphSet produced segmentation faults due to null pointer dereferencing in Template Q3 and several invalid writes when accessing already freed memory.

TryAlgo In June 2020, the authors of the *TryAlgo* implementation noted on their website that they "have problems implementing this data structure, and cannot provide at this point a correct implementation in tryalgo". Furthermore, they note that "the current implementation has a complexity in the order of $n * m$, however an implementation in $O(n + m + s)$ is possible". As we thus assumed their implementation to be neither correct nor linear-time, we excluded it from our evaluation.

SageMath The main routine set_contiguous of the PQ-tree of *SageMath* recursively traverses the tree starting from its root as follows: It first calls set_contiguous recursively on all children of the current node, then calls flatten, calls set_contiguous recursively on all children again and then proceeds to sort the children depending on whether they are full, partial, or empty. The flatten function for removing degree-2 nodes is implemented to recurse itself on all children in the subtree, making the running time of set_contiguous quadratic in the tree size. We modified the implementation to only flatten the current level and dropped the second recursive call to set_contiguous, improving the running time to linear in the tree size without generating incorrect results.

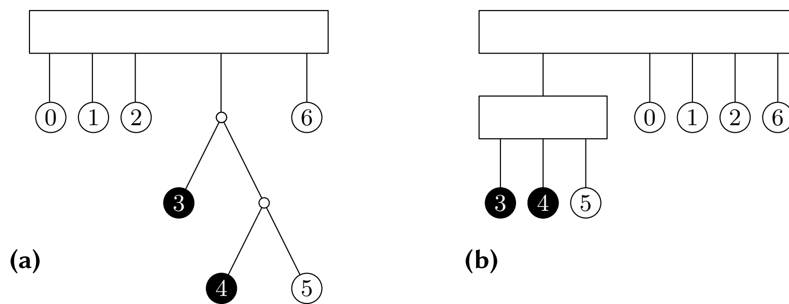


Figure 8.11: (a) The PQR-tree [0 1 2 (3 (4 5)) 6] not containing any R-nodes with its root Q-node depicted as rectangle and the two P-nodes depicted as small circles. **(b)** The result of Zanetti's implementation applying the restriction {3,4} to the former tree, the tree [[3 4 5] 0 1 2 6] which clearly represents a different set of restrictions.

Zanetti We found that *Zanetti*'s data structures became inconsistent after some restrictions, which was also already independently reported on GitHub.[15] This happened mostly after restrictions having a terminal path length of greater than 1. As the restrictions generated when serializing a PC-tree only have very short terminal paths and the inconsistency is usually only found when modifying the same area of the tree again, only few of these cases surfaced in our tests on SER-POS. Only when applying multiple bigger restrictions consecutively, these issues surfaced more often, i.e., at some point during the planarity test for close to all graphs with $m = 2n$ and also some of the graphs with $m = 3n - 6$. We also found a second, independent issue, where Zanetti's implementation generates C-nodes with their children in the wrong order. An example where this happens is shown in Figure 8.11.

As Zanetti's Java implementation still has a very good running time in practice, we decided to port its Java code to C++ to be able to perform a direct comparison with the other C++ implementations. As the implementation uses almost no Java-specific features, the conversion mostly involved replacing Java Object variables with C pointers and Java utility classes with their C++ stdlib equivalents. The only non-trivial change was that, because Zanetti stores the Union-Find information directly in the nodes and not in an external array, we had to implement reference counting for Zanetti's tree nodes to ensure that the lifetime of nodes which are no longer part of the tree, but still referenced in the Union-Find data structure, is handled properly. We made sure that both the Java and the C++ version not only produced equivalent

15 https://github.com/jppzanetti/PQRTree/issues/2

output, but actually keep the same PQR-tree state in memory. Where both implementations differ is that Java immediately reports inconsistencies of the data structure, e.g., by throwing a NullPointerException, whereas the SIGSEGFAULT of C++ might not be immediately triggered. This generates a few more data points with an invalid result, where the Java implementation already crashed.

8.3.3 Results

Our experiments turn out that SageMath, even with the improvements mentioned above, is on average 30 to 100 times slower than all other implementations.[16] For the sake of readability, we scale our plots to focus on the other implementations. As the main application of PC-/PQ-trees is applying possible restrictions, we first evaluate on the dataset SER-POS. Figure 8.12 shows the running time for individual restrictions based on the size of the restriction (i.e., the number of full leaves) and the overall size of the tree. Figure 8.12 (a) clearly shows that for all implementations the running time is linear in the size of the restriction. Figure 8.12 (b) suggests that the running time of Reisle and GraphSet does not solely depend on the restriction size, but also on the size of the tree. To verify this, we created for each implementation a heatmap that indicates the average running time depending on both the tree size and the restriction size, shown in Figure 8.13 (a). The diagonal pattern shown by SageMath, Reisle, and GraphSet confirms the dependency on the tree size. All other implementations exhibit vertical stripes, which shows that their running time does not depend on the tree size. Finally, Figure 8.13 (b) shows the running time compared to the terminal path length. As expected, all implementations show a linear dependency on the terminal path length, with comparable results to Figure 8.12 (a).

Figure 8.14 (a) shows the performance on the dataset MAT-POS depending on the restriction size. The ranking of the implementations is similar to the results on SER-POS; only OGDF performs noticeably worse on MAT-POS. Note that Figure 8.8 (d) shows that this dataset contains more larger restrictions, which effectively result in rather short terminal paths (see Figures 8.9 (a) and 8.9 (d)). As the rows get darker towards their end in Figure 8.8 (d), the restrictions originating from the consecutive ones matrices exhibit a correlation between restriction size and tree size. Thus, the running time of every implementation would exhibit a linear dependence on the tree size on this dataset. We therefore focus on the test cases generated using the planarity test for a more detailed analysis of the performance.

16 Part of this might be due to the overhead of running the code with CPython. As the following analysis shows, SageMath also has other issues, allowing us to safely exclude it.

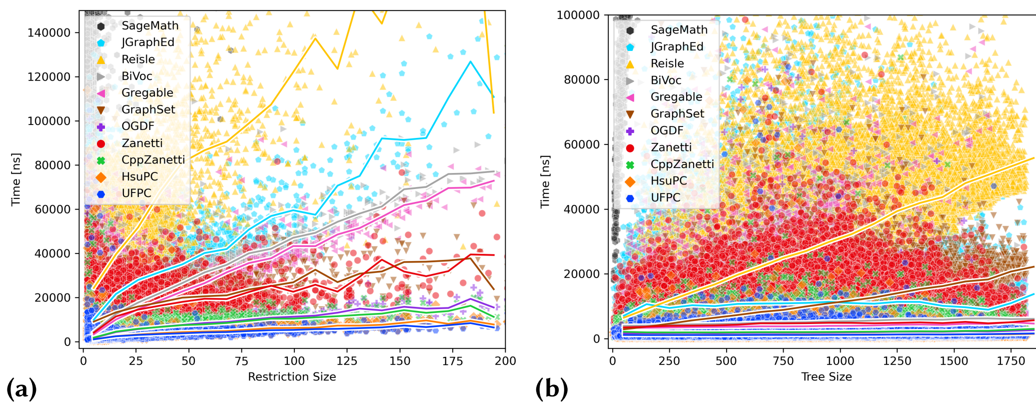


Figure 8.12: Running time for SER-POS restrictions depending on **(a)** restriction size and **(b)** tree size. The solid lines show the arithmetic mean for the respective implementation. Note the different scales on the y-axis.

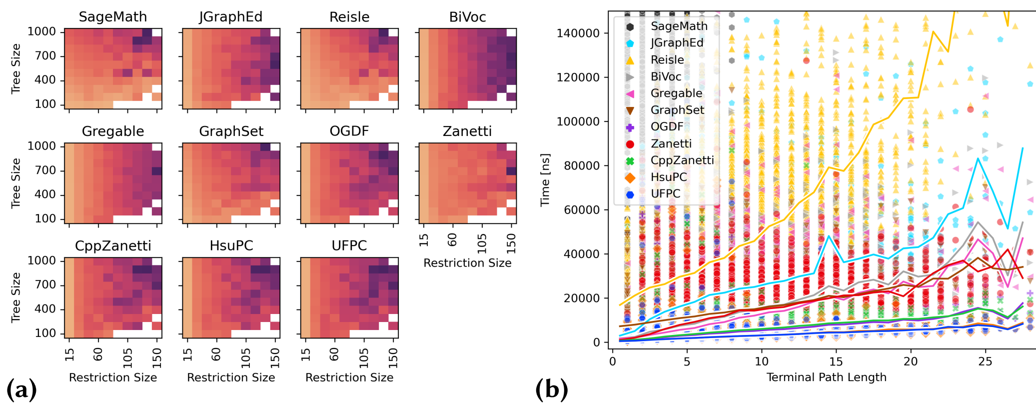


Figure 8.13: (a) A heatmap showing the average running time of SER-POS restrictions, depending on both the size of the restriction and the size of the tree. The color scale is based on the maximum running time of each respective implementation, where darker colors indicate longer running times. **(b)** Running time for SER-POS restrictions depending on the terminal path length.

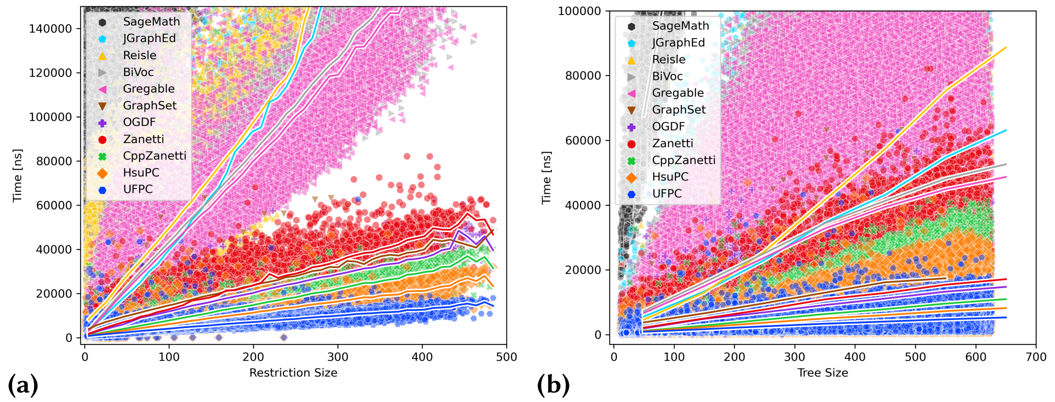


Figure 8.14: Running time for MAT-POS restrictions depending on **(a)** restriction size and **(b)** tree size for all implementations.

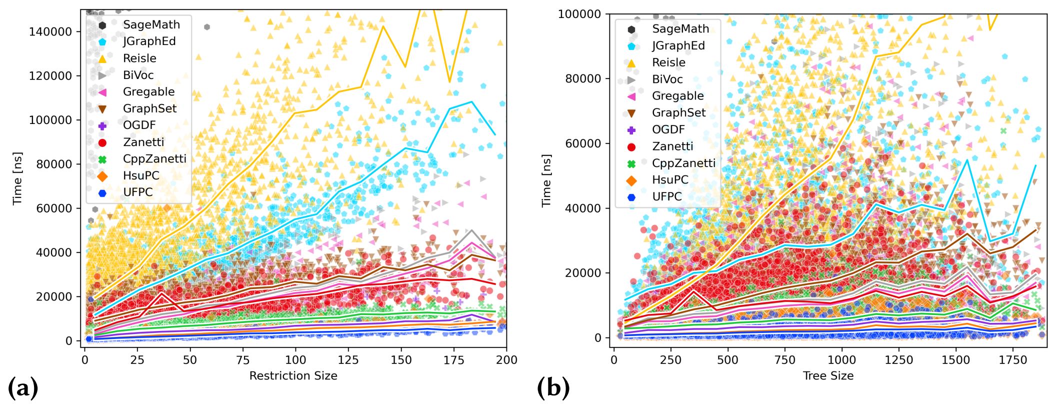


Figure 8.15: Running time for SER-IMP restrictions depending on **(a)** restriction size and **(b)** tree size for all implementations.

Figure 8.15 shows the performance on the dataset SER-IMP. The performance is comparable with that on SER-POS. Noteworthy is that Zanetti performs quite a bit worse, which is due to its implementation not being able to detect failure during a labeling step. It always performs updates until a so-called R-node would be generated. Altogether, the data from SER-POS and MAT-POS shows that the implementations GraphSet, OGDF, Zanetti, HsuPC and UFPC are clearly superior to the others. In the following, we conduct a more detailed comparison of these implementations by integrating them into a planarity test and running them on much larger instances, i.e., the data set DIR-PLAN. In addition to an update method, this requires a method for replacing the now-consecutive leaves by a P-node with a given number of child leaves. Adding the necessary functionality would be a major effort for most of the implementations, which is why we only adapted the most efficient implementations to run this set. We also exclude GraphSet from this experiment; the fact that it scales linearly with the tree size causes the planarity test to run in quadratic time (see also Section 8.3.2). Figure 8.16 again shows the running time of individual restrictions depending on the restriction size. Curiously, Zanetti produces incorrect results for nearly all graphs with $m = 2n$ in Figure 8.16 (a). As the initial tests already showed, the implementation has multiple flaws; one major problem is already described in an issue on GitHub, while we give a small example of another independent error in Figure 8.11. Both plots show that HsuPC is more than twice as fast as OGDF and that UFPC is again close to two times faster than HsuPC. Zanetti's running time is roughly the same as that of HsuPC, while converting its Java code to C++ brings the running time down close to that of UFPC.

As OGDF is the slowest, we use it as baseline to calculate the speed-up of the other implementations. Figure 8.17 (a) shows that the running time improvement for all three implementations is the smallest for small restrictions, quickly increasing to the final values of roughly 0.4 times the running time of OGDF for HsuPC and 0.25 for both CppZanetti and UFPC. Figure 8.17 (b) shows the speed-up depending on the length of the terminal path. For very short terminal paths (which are common in our datasets), both implementations are again close; but already for slightly longer terminal paths UFPC quickly speeds up to being roughly 20 % faster than CppZanetti. This might be because creating the central node in step (4a) is more complicated for UFPC, as the data structure without edge objects does not allow arbitrarily adding and removing edges (which is easier for HsuPC) and allowing cyclic restrictions forces UFPC to also pay attention to various special cases (which are not necessary for PQ-trees).

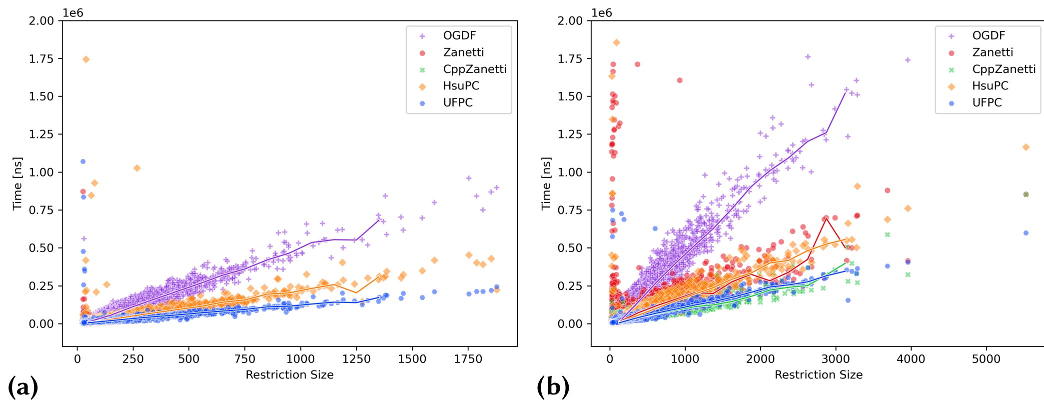


Figure 8.16: Running time of individual restrictions of DIR-PLAN with OGDF, Zanetti and our implementations for graphs of size **(a)** $m = 2n$ and **(b)** $m = 3n - 6$. Please note the different scales on the x-axis.

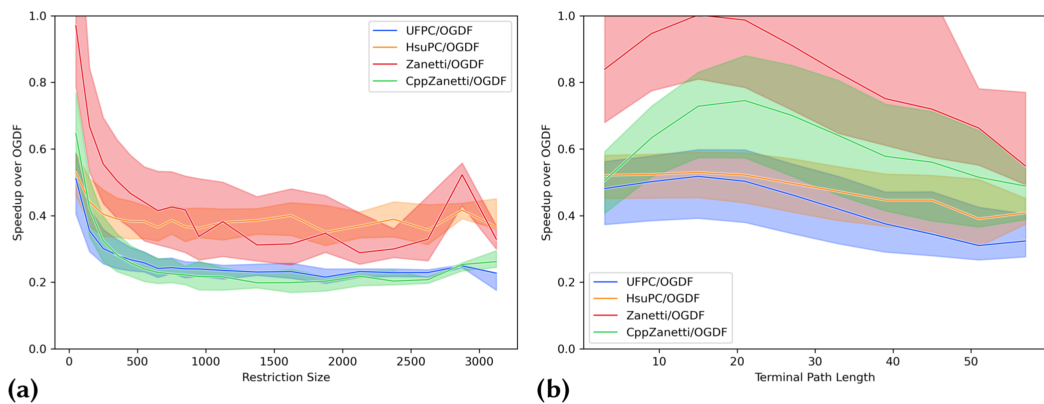


Figure 8.17: Median performance increase depending on **(a)** the size of the restriction and **(b)** the terminal path length, with OGDF as baseline. The shaded areas show the interquartile range.

8.4 Testing Planarity and Generating Embeddings

To put the 4-fold speed-up of UFPC over OGDF into context, we compared the OGDF implementations of the planarity test by Booth and Lueker and the one by Boyer and Myrvold on our graph instances. The OGDF Boyer and Myrvold implementation was roughly 50 % faster than the one based on Booth and Lueker's algorithm. Replacing the PQ-trees, which are the core part of the latter, by an implementation that is 4 times faster, might make this planarity test run faster than the one by Boyer and Myrvold. In order to make a fair comparison between the different planarity tests, we also need to take into account the embedding generation, which our PC-tree based planarity test not yet provides.

Chiba et al. [Chi+85] show how an embedding can be constructed along a run of the planarity test by Booth and Lueker; see Theorem 4.3. Recall that the planarity test by Booth and Lueker requires a decomposition into biconnected components and works along an st-ordering (see Chapter 4). To avoid the practical overhead of computing both, we instead prefer to implement the more recent generalization of the planarity test by Haeupler and Tarjan [HT08]. This test works along a reversed DFS-order, which is easier to compute than an st-ordering, and can also handle non-biconnected components, alleviating the need for a prior decomposition into biconnected components. In return for this advantages, the algorithm is slightly more complicated and now also needs to allow for merging multiple PC-trees at an inserted vertex (see Section 5.1.4). Unfortunately, the embedder by Chiba et al. [Chi+85] is not directly able to handle this [Fre22]. Instead, we turn to an approach for generating embeddings from an entirely different family of planarity tests.

The left-right planarity test as described by Brandes [Bra09] also works along a DFS-tree, but processes individual edges instead of inserting whole vertices. The test labels each back-edge as “left” or “right” depending on whether it returns to the tree in respectively counter-clockwise or clockwise direction in an implicitly generated planar embedding. Note that storing the order of incoming back-edges in relation to their outgoing tree-edge as done by Chiba et al. [Chi+85] in the Booth and Lueker planarity test also implicitly yields such labeling, depending on whether a back-edge appears before or after its tree-edge in the stored order [Feu23]. Even more, these orders obtained from PC-trees are equivalent to the LR Ordering described by Brandes [Bra09, Definition 7]. Based on these stored orders (after ensuring their flips are consistent [Chi+85; Fre22]), we can thus use the approach described by Brandes to obtain a planar embedding from an LR Ordering [Bra09, Lemma 8 and Algorithm 6].

Based on these observations, we implement our UFPC based embedder as follows. We implement the planarity test as suggested by Haeupler and Tarjan [HT08]. We store and track the flips of orders of consecutive edges as described by

Chiba et al. [Chi+85] independently for each PC-tree, using the implementation described by Frey [Fre22]. Instead of incrementally constructing an embedding from the obtained LR Ordering as suggested by Brandes [Bra09, Algorithm 6], we directly generate the embedding by sorting the adjacency lists using an appropriate comparison function based on the positions in the LR Ordering together with a linear-time radix sort. See the work by Feulner [Feu23] for details on this.

Set-Up

We build upon the evaluation by Feulner [Feu23] comparing this Haeupler-Tarjan-style planarity test and embedder with the Boyer-Myrvold implementation of the OGDF [BM99; Boy+04a; Chi+14]. We also add the PQ-tree-based Booth-Lueker implementation [BL76] from the OGDF to the comparison. Additionally, we compare a variant of our Haeupler-Tarjan implementation, where instead of a global radix sort that generates the embedding, we use one call to the C++ `std::sort` method for each vertex. Note that this increases the asymptotic running time from linear to $O(n + \Delta \log \Delta)$, where Δ is the maximum vertex degree. We run these comparisons on random connected, biconnected and triconnected graphs with $n \in \{100\,000, 200\,000, \dots, 1\,000\,000\}$ vertices and $m \in \{1.5n, 2n, 2.5n, 3n - 6\}$ edges, generating 5 instances via the respective OGDF generator for each such combination. For each combination of parameters, we also generate 5 random connected but not biconnected graphs with a prescribed number $b \in \{5, 10, 25, 50, 100, 250, 500, 1000\}$ of blocks. In this instance set, the size of each block is also chosen randomly, which leads to these graphs having slightly fewer than the aimed-for number of edges and a higher variation of densities.

Results

Figure 8.18 (a) shows the clearly-linear absolute running times of the OGDF implementation of the state-of-the-art Boyer-Myrvold (BM) embedder on our sets of (bi-/tri-)connected graphs. In the relation to this, the remaining plots of Figure 8.18 show the relative running times of the other algorithms. The Booth-Lueker algorithm (BL) is clearly slower than the Boyer-Myrvold, with a visible additional overhead when also generating an embedding. This overhead is not visible when generating an embedding using the Boyer-Myrvold implementation, indicating that it always generates an embedding even when this is not requested. The speed-up of our Haeupler-Tarjan (HT) embedder implementation over the Boyer-Myrvold one ranges between one quarter and one third, depending on the density of the graph. Using the super-linear `std::sort` instead of radix sort for HT seems to be very slightly slower.

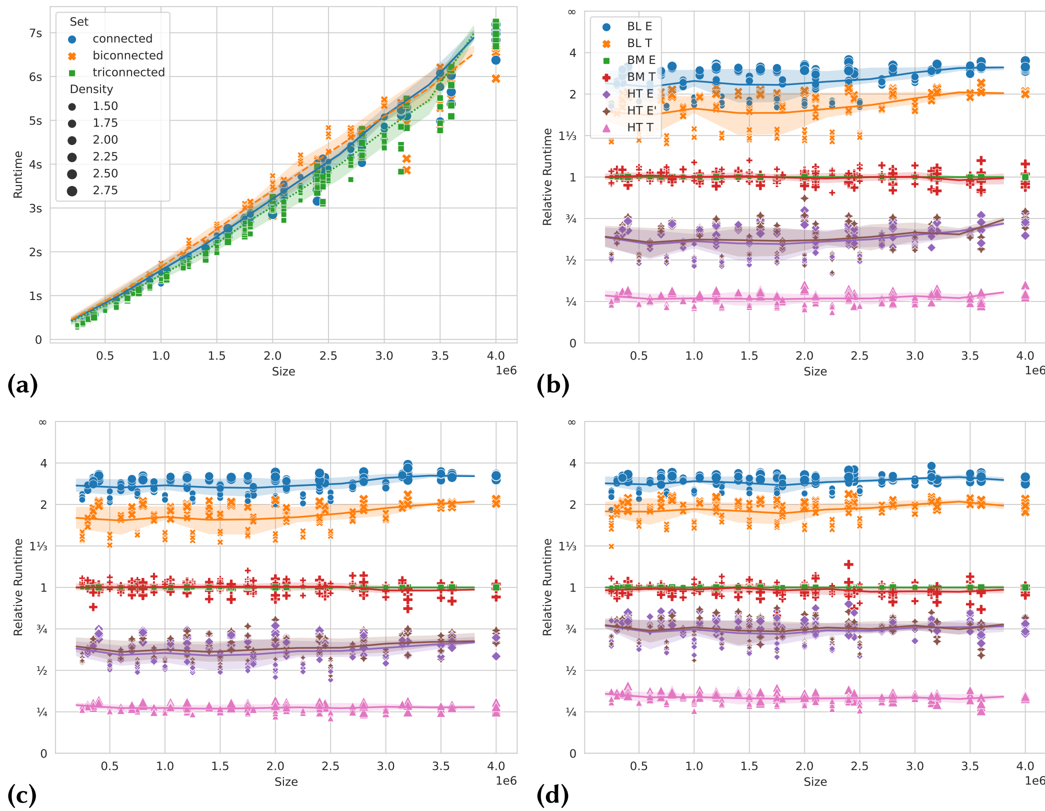


Figure 8.18: (a) Running time of the OGDF Boyer-Myrvold embedder BM E on our sets of (bi-/tri-)connected graphs. The remaining plots show the running time of all algorithms compared to that of BM E on the set of **(b)** connected, **(c)** biconnected, and **(d)** triconnected graphs. Algorithms are abbreviated by the authors' initials, appending T for planarity tests and E for embedders. Algorithm HT E' uses the super-linear std::sort instead of radix sort. The x-axis denotes the graph size as sum of numbers of vertices and edges, lines show medians, shaded areas show interquartile ranges.

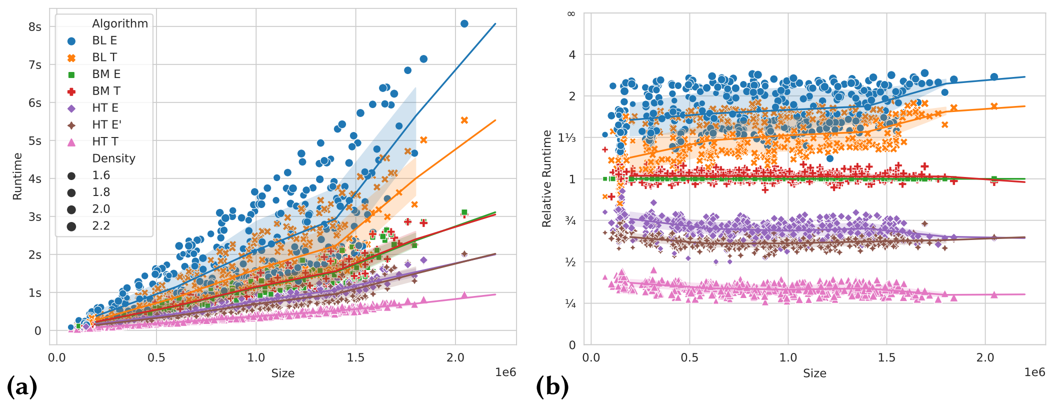


Figure 8.19: **(a)** Absolute and **(b)** relative running time of all algorithms on the set of connected but not biconnected graphs.

Figure 8.19 shows the running times on the remaining set of connected but not biconnected graphs. Here, the random graph generator chooses a random size for each block, using the requested sizes only as an upper bound and yielding more diverse albeit slightly smaller graph sizes. This also leads to the relative running times varying more, although they all still clearly follow the same trend as in the other data sets. Interestingly, the super-linear HT E' now seems to be faster than the radix-sort-based linear HT E.

All three implementations correctly classified all graphs we tested. To guard against overly-permissive implementations yielding false-positive results, we also tested them on further random non-planar graphs. Although the running time here strongly depends on the order in which the graphs is processed, our implementation is consistently much faster than BM on these graphs; see Figure 8.20. This might be in parts due to BM being able to report Kuratowski subdivisions for non-planar graphs [CMS07].

To summarize, this short evaluation shows that our planarity test implementation based on the Haeupler-Tarjan algorithm is clearly the fastest among the three OGDF-based implementations we compared. Using a slight variation of this approach with a super-linear asymptotic running time yields faster practical running times on one out of four evaluated graph sets. This shows that there may still be some speed-up to gain from further engineering this approach. Still, to obtain decisive updated ranking, one would also need to take into account other potentially more efficient implementations of these algorithms (see for example [Boy+04a; FMR06]) and a much larger set of test instances taken from more, different sources. We leave such a much larger in scale evaluation for future work.

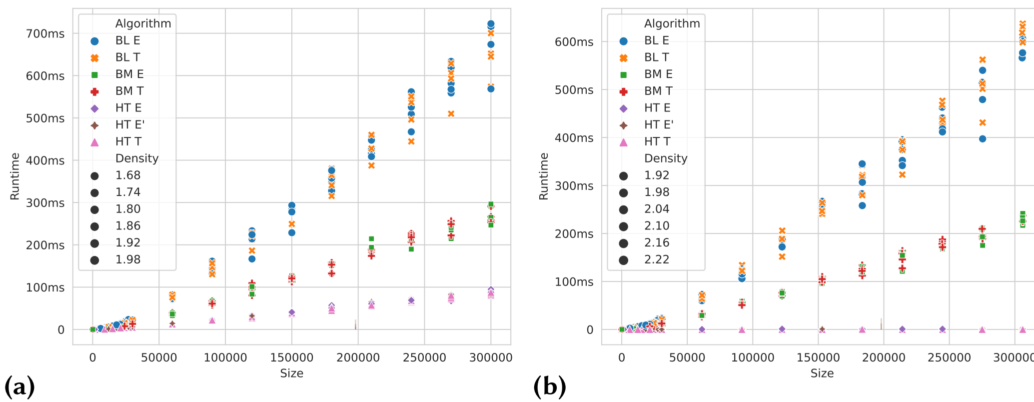


Figure 8.20: Absolute running time of all algorithms on sets of random non-planar biconnected **(a)** and triconnected **(b)** graphs.

8.5 Conclusion

In this chapter we have presented the first fully generic and correct implementations of PC-trees. One implementation follows the original description of Hsu and McConnell [HM03; HM04], which contains several subtle mistakes in the description of the labeling and the computation of the terminal path. This may be the reason why no fully generic implementation has been available so far. We give a corrected version that also includes several small simplifications.

Furthermore, we provided a second, alternative implementation, using Union-Find [TL84] to replace many of the complications of Hsu and McConnell's original approach. Technically, this increases the running time to $O((|R|+p)\cdot\alpha(|L|))$, where α is the inverse Ackerman function. In contrast, our evaluations show that the Union-Find-based approach is even faster in practice, despite the worse asymptotic running time.

Our experimental evaluation with a variety of other implementations reveals that surprisingly few of them are fully correct. Only two other implementation correctly handle all our test cases. The fastest of them is the PQ-tree implementation of OGDF, which our Union-Find-based PC-tree implementation beats by roughly a factor of 4. Interestingly, the Java implementation of PQR-trees by Zanetti achieves a similar speed-up once ported to C++. However, Zanetti's Java implementation is far from correct and it is hard to say whether it is possible to fix it without compromising its performance.

Altogether, our results show that PC-trees are not only conceptually simpler than PQ-trees but also perform well in practice, especially when combined with

Union-Find. Based on this, we implemented a planarity test and embedder following the approach by Haeupler and Tarjan [HT08]. In our short evaluation, this implementation is one quarter to one third faster than the OGDF implementation of the state-of-the art algorithm by Boyer and Myrvold [BM99]. This shows that, roughly 20 years after this state-of-art was established [Boy+04a], the ranking of planarity testing algorithms is in need of a re-evaluation. We leave such a detailed evaluation, also comparing other potentially more efficient implementations and a larger set of test instances, for future work.

Another direction for future work are PC-tree-based tests for constrained planarity variants. Our implementation of PC-trees can be used as base for implementing the Level Planarity test by Brückner and Rutter [Brü21, Section 5] as well as the Partially Embedded Planarity we present in Chapter 5.

9 Engineering the Synchronized Planarity Algorithm

This chapter is based on joint work with Ignaz Rutter, which appeared at ALENEX 2024 [7]. Our source code is available at github.com/N-Coder/syncplan and also archived at softwareheritage.org as swh:1:snp:0dae4960cc1303cc3575cf04924e19d664f8ad87.

The problem Synchronized Planarity not only generalizes many constrained planarity variants, among them in particular Level and Clustered Planarity as well as variants of SEFE, but also has a comparatively simple quadratic-time solution, which is discussed in Chapter 6. Akin to the Goldberg and Tarjan push-relabel algorithm [GT88], it uses few and simple operations that can be applied in arbitrary order. Through reductions from many other problems (see Figures 3.1 and 9.1 for an overview), an implementation would also allow to solve other constrained planarity problems for which no practical solution is available. This wide area of possible applications and the fact that the algorithm offers several degrees of freedom make it an ideal starting point for algorithm engineering.

In this chapter, we describe our implementation of the Synchronized Planarity algorithm, which we evaluate by comparing its results and running times to those of two existing implementations for the Clustered Planarity problem. We complement the theoretical running time analysis from Chapter 6 with practical measurements, highlighting which parts of the algorithm take the most time. Based on this, we engineer the algorithm by analyzing how to best employ the degrees of freedom present in the algorithm and by proposing algorithmic improvements to overcome performance bottlenecks. Section 9.1 gives an overview of previous practical approaches to constrained planarity problems. In Section 9.2 we describe our implementation of Synchronized Planarity and evaluate its performance in comparison with the two other available Clustered Planarity implementations. We tune the running time of our implementation to make it practical on even large instances in Section 9.3. We analyze the effects of our engineering in greater detail in Section 9.4.

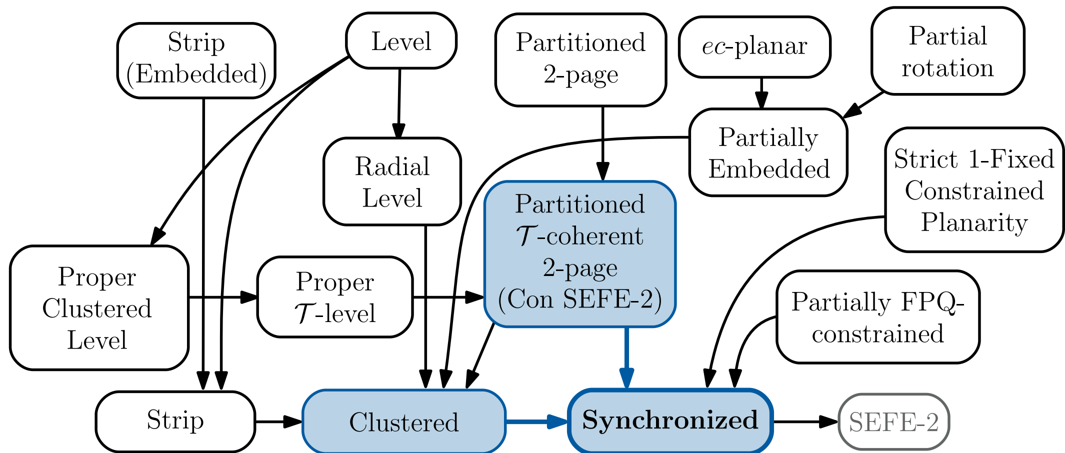


Figure 9.1: Constrained planarity variants related to Synchronized Planarity, updated selection from [DaL15]. Problems and reductions marked in blue are used for generating test instances.

9.1 Related Work

Surprisingly, in contrast to their intense theoretical consideration, constrained planarity problems have only received little practical attention so far. Of all variants, practical approaches to Clustered Planarity were studied the most, although all implementations predate the first generic polynomial-time solution and thus either have an exponential worst-case running time or cannot solve all instances. Chimani et al. [Chi+08] studied the problem of finding maximal clustered-planar subgraphs in practice using an Integer Linear Program (ILP) together with a branch-and-cut algorithm. A later work [CK12] strengthened the ILP for the special case of testing Clustered Planarity, further improving the practical running time. The work by Gutwenger et al. [GMS14] takes a different approach by using a Hanani-Tutte-style formulation of the problem based on the work by Schaefer [Sch13]. Unfortunately, their polynomial-time testing algorithm cannot solve all instances and declines to make a decision for some instances. The Hanani-Tutte-approach solved instances with up to 60 vertices and 8 clusters in up to half a minute, while the ILP approach only solves roughly 90 % of these instances within 10 minutes [GMS14].

The only other constrained planarity variant for which we could find experimental results is Partitioned 2-page Book Embedding. Angelini et al. [ABD12] describe an implementation of the SPQR-tree-based linear-time algorithm by Hong and Nagamochi [HN09], which solves instances with up to 100 000 vertices and two clusters in up to 40 seconds. Unfortunately, their implementation is not pub-

licly available. For (Radial) Level Planarity, prototypical implementations were described in the dissertations by Leipert [Lei98] and Bachmaier [Bac04], although in both cases neither further information, experimental results, nor source code is available. The lack of an accessible and correct linear-time implementation may be due to the high complexity of the linear-time algorithms [Brü21]. Simpler algorithms with a super-linear running time have been proposed [Ful+12; HH07; Ran+01]. For these, we could only find an implementation by Estrella-Balderrama et al. [EFK10] for the quadratic algorithm by Harrigan and Healy [HH07]. Unfortunately, this implementation has not been evaluated experimentally and we were also unable to make it usable independently of its Microsoft Foundation Classes GUI, with which it is tightly intertwined.

We are not aware of further practical approaches for constrained planarity variants. Note that while the problems Partitioned 2-page Book Embedding and Level Planarity have linear-time solutions, they are much more restricted than Synchronized Planarity (see Figure 9.1) and have no usable implementations available. We thus focus our comparison on solutions to the Clustered Planarity problem which, besides being a common generalization of both other problems, fortunately also has all relevant implementations available.

Dataset	#	Vertices	Density	Components	Clusters/Pipes	d
C-OLD	1643	≤59 (17.2)	0.9–2.2 (1.4)	=1	≤19 (4.2)	≤256 (34.0)
C-NCP	13 834	≤500 (236.8)	0.6–2.9 (1.9)	≤48 (21.7)	≤50 (16.8)	≤5390 (783.3)
C-MED	5171	$\leq 10^3$ (311.6)	0.9–2.9 (2.3)	≤10 (5.1)	≤53 (16.1)	≤7221 (831.8)
C-LRG	5096	$\leq 10^5$ (15 214.1)	0.5–3.0 (2.4)	≤100 (29.8)	≤989 (98.8)	≤2 380 013 (44 788.7)
SEFE-LRG	1008	$\leq 10^4$ (3800.0)	1.1–2.4 (1.7)	=1	≤20 000 (7600.0)	≤113 608 (34 762.4)
SP-LRG	1587	$\leq 10^5$ (25 496.6)	1.3–2.5 (2.0)	≤100 (34.5)	≤20 000 (1467.4)	≤139 883 (9627.5)

Table 9.1: Statistics for our different datasets, values in parentheses are averages. Column # shows the number of instances while column d shows the total number of cluster-border edge crossings or the total degree of all pipes, depending on the underlying instances.

	C-OLD				C-NCP				C-MED			
	ILP	HT	HT-f	SP[d]	ILP	HT	HT-f	SP[d]	ILP	HT	HT-f	SP[d]
Y	732	792	792	792	181	1327	1534	1535	953	762	2696	5170
N	800	851	851	851	946	6465	6463	12 308	0	85	85	0
ERR	0	0	0	0	5214	0	0	0	1263	0	0	0
TO	111	0	0	0	7502	6051	5846	0	2955	4324	2390	1

Table 9.2: Counts of the results ‘yes’, ‘no’, ‘error’, and ‘timed out’ on C-OLD, C-NCP and C-MED.

9.2 Clustered Planarity in Practice

In this section, we shortly describe our C++ implementation of the Synchronized Planarity algorithm from Chapter 6 and compare its running time and results on instances derived from Clustered Planarity with those of the two existing implementations by Chimani et al. [Chi+08; CK12] and by Gutwenger et al. [GMS14]. We base our implementation on the graph data structures provided by the OGDF [Chi+14] and, as only other dependency, use the PC-tree implementation `UFPC` described in the previous Chapter 8 for the embedding trees.

We consider a pipe *feasible* if it can be removed by applying any one of the three operations. The algorithm for Synchronized Planarity makes no restriction on how the next feasible pipe should be chosen. Moreover, it can be shown that if a pipe is not feasible, then this is directly caused by a close-by pipe with endpoints of higher degree (see Lemma 6.7 in Section 6.3.6). For now, we will use a heap to always use a pipe of maximum degree that is thus feasible.

The operations used for solving Synchronized Planarity heavily rely on (bi-)connectivity information while also making changes to the graph that may affect this information. As recomputing the information before each step would pose a high overhead, we maintain this information in the form of a BC-forest (i.e. a collection of BC-trees). To generate the embedding trees needed by the `PropagatePQ` and `SimplifyMatching` operations, we implement the Booth-Lueker algorithm for testing planarity [BL76; Pat13] using PC-trees. We use that, after processing all vertices of a biconnected component, the resulting PC-tree corresponds to the embedding tree of the vertex that was processed last; see Section 4.2.

9.2.1 Evaluation Set-Up

We compare our implementation of Synchronized Planarity with the Clustered Planarity implementations `ILP` by Chimani et al. [Chi+08; CK12] and `HT` by Gutwenger at al. [GMS14] (not to be confused with the Haeupler-Tarjan planarity test from Section 8.4). Both are written in C++ and are part of the OGDF. The `ILP` implementation by Chimani et al. [Chi+08; CK12] uses the ABACUS ILP solver [Elf+01] provided with the OGDF. We refer to our Synchronized Planarity implementation processing pipes in descending order of their degree as `SP[d]`. We use the embedding it generates for yes-instances as certificate to validate all positive answers. For the Hanani-Tutte algorithm, we give the running times for the modes with embedding generation and verification (`HT`) and the one without (`HT-f`) separately. Note that `HT-f` only checks an important necessary, but not sufficient condition and thus may falsely classify negative instances as positive, see [GMS14,

Figure 3] and [Ful+15, Figure 16] for examples where this is the case. Variant HT tries to verify a positive answer by generating an embedding, which works by incrementally fixing parts of a partial embedding and subsequently re-running the test. This process may fail at any point, in which case the algorithm can make no statement about whether the instance is positive or negative [GMS14, Section 3.3]. We note that, in any of our datasets, we neither found a case of HT-f yielding a false-positive result nor a case of a HT verification failing. The asymptotic running time of HT-f is bounded by $O(n^6)$ and the additional verification of HT adds a further factor of n [GMS14].

We combine the Clustered Planarity datasets that were previously used for evaluations on HT and ILP to form the set C-OLD [Chi+08; CK12; GMS14]. We apply the preprocessing rules of Gutwenger at al. [GMS14] to all instances and discard instances that become trivial, non-planar or cluster-connected, since the latter are easy to solve [Cor+08]. This leaves 1643 instances; see Table 9.1. To create the larger dataset C-NCP, we used existing methods from the OGDF to generate instances with up to 500 vertices and up to 50 clusters. This yields 15 750 instances, 13 834 out of which are non-trivial after preprocessing. As this dataset turned out to contain only 10 % yes-instances, we implemented a new clustered-planar instance generator that is guaranteed to yield yes-instances. We use it on random planar graphs with up to 1000 vertices to generate 6300 clustered-planar instances with up to 50 clusters. Out of these, 5171 are non-trivial after preprocessing and make up our dataset C-MED. We provide full details on the generation of our dataset at the end of this section.

We run our experiments on Intel Xeon E5-2690v2 CPUs (3.00 GHz, 10 Cores, 25 MB Cache) with a memory usage limit of 6 GB. As all implementations are single-threaded, we run multiple experiments in parallel using one core per experiment. This allows us to test more instances while causing a small overhead which affects all implementations in the same way. The machines run Debian 11 with a 5.10 Linux Kernel. All binaries are compiled statically using gcc 10.2.1 with flags `-O3 -march=native` and link-time optimization enabled. We link against slightly modified versions of OGDF 2022.02 and the PC-tree implementation UFPC described in Chapter 8. The source code of our implementation and all modifications are available at github.com/N-Coder/syncplan,[17] while our dataset is on Zenodo with DOI 10.5281/zenodo.7896021.

17 It is also archived at Software Heritage with ID swh:1:snp:0dae4960cc1303cc3575cf04924e19d664f8ad87.

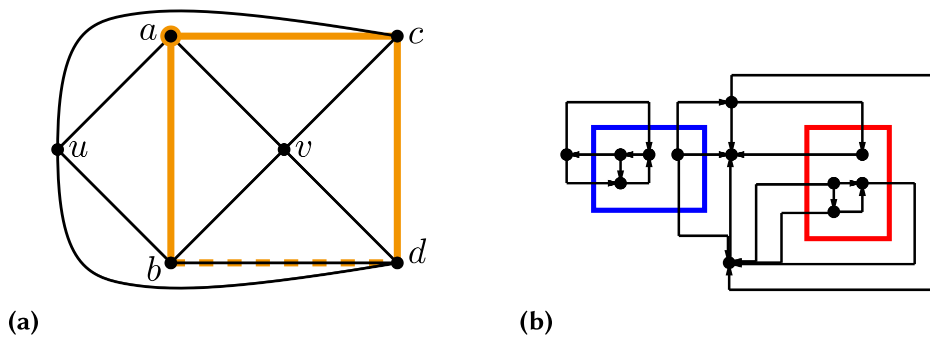


Figure 9.2: (a) Converting the subtree $\{a, b, c, d\}$ with root a (shown in orange) into a cluster will separate vertices u and v, as the edge bd (dashed) will also be part of the cluster. **(b)** A clustered-planar graph with two clusters (in addition to the root cluster) that HT classifies as "nonCPlanarVerified".

Details on Dataset Generation

The dataset C-OLD is comprised of the datasets P-Small, P-Medium, P-Large by Chimani and Klein [CK12] together with PlanarSmallR (a version of PlanarSmall [Chi+08] with preprocessing applied), PlanarMediumR and PlanarLargeR by Gutwenger et al. [GMS14]. The preprocessing reduced the dataset of Chimani and Klein [CK12] to 64 non-trivial instances, leading to dataset C-OLD containing 1643 instances in total.

The OGDF library can generate an entirely random clustering by selecting random subsets of vertices. It can also generate a random clustered-planar and cluster-connected clustering on a given graph by running a depth-first search that is stopped at random vertices, forming new clusters out of the discovered trees. To generate non-cluster-connected but clustered-planar instances, we temporarily add the edges necessary to make a disconnected input graph connected. For the underlying graphs of C-NCP, we use the OGDF to generate three instances for each combination of $n \in \{100, 200, 300, 400, 500\}$ nodes, $m \in \{n, 1.5n, 2n, 2.5n, 3n - 6\}$ edges, and $d \in \{10, 20, 30, 40, 50\}$ distinct connected components. For each input graph, we generate six different clusterings, three entirely random and three random clustered-planar, with $c \in \{3, 5, 10, 20, 30, 40, 50\}$ clusters. This yields 15 750 instances, 13 834 out of which are non-trivial after preprocessing.

It turns out that roughly 90 % of these instances are not clustered-planar (see Table 9.2), even though half of them are generated by a method claiming to only generate clustered-planar instances. This is because the random DFS-subtree used for clusters by the OGDF only ensures that the generated cluster itself, but not its complement are connected. Thus, if the subgraph induced by the selected vertices

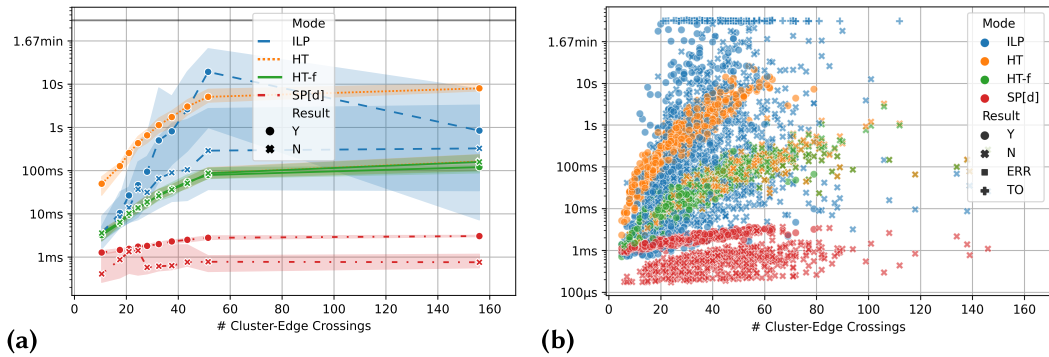


Figure 9.3: Median running times on dataset C-OLD **(a)** together with the underlying scatter plot **(b)**. For each algorithm, we show running times for yes- and no-instances separately. Markers show medians of bins each containing 10 % of the instances. Shaded regions around each line show inter-quartile ranges.

contains a cycle, this cycle may separate the outside of the cluster; see Figure 9.2 (a). To reliably generate yes-instances, we implemented a third method for generating random clusterings. We first add temporary edges to connect and triangulate the given input graph. Afterwards, we also generate a random subtree and contract it into a cluster. Each visited vertex is added to the tree with a probability set according to the desired number of vertices per cluster. To ensure the non-tree vertices remain connected, we only add vertices to the tree whose contraction leaves the graph triangulated, i.e., that have at most two neighbors that are already selected for the tree. We convert the selected random subtrees into clusters and contract them for the next iterations until all vertices have been added to a cluster.

As we do not need multiple connected components to ensure the instance is not cluster-connected for our Clustered Planarity instance generator, we used fewer steps for the corresponding parameter, but extended the number of nodes up to 1000 for C-MED. The underlying graphs are thus comprised of three instances for each combination of $1 \leq n \leq 1000$ nodes with $0 \equiv n \mod 100$ (i.e. n is a multiple of 100), $m \in \{n, 1.5n, 2n, 2.5n, 3n - 6\}$ edges, and $d \in \{1, 10, 25, 50\}$ distinct connected components. For each input graph, we generate three random clustered-planar clusterings with an expected number of $c \in \{3, 5, 10, 20, 30, 40, 50\}$ clusters. This yields 6300 instances which are guaranteed to be clustered-planar, 5171 out of which are non-trivial after preprocessing and make up our dataset C-MED.

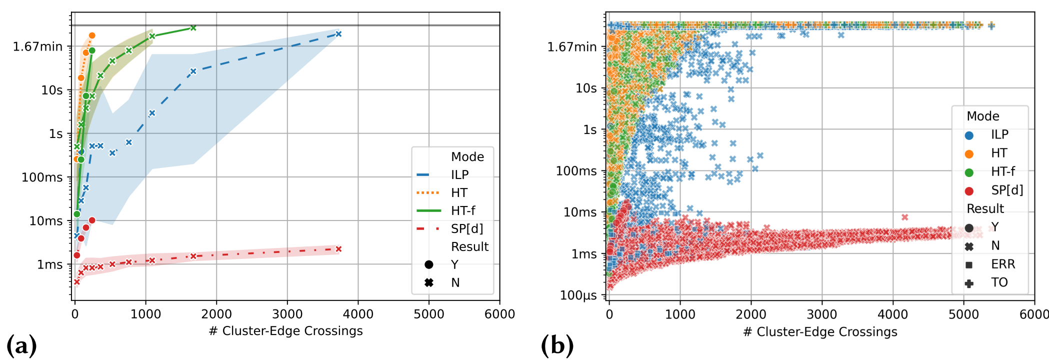


Figure 9.4: Median running times **(a)** and scatter plot **(b)** on dataset C-NCP.

9.2.2 Results

Table 9.2 shows the results of running the different algorithms. The dataset C-OLD is split in roughly equal halves between yes- and no-instances and all algorithms yield the same results, except for the 111 instances for which the ILP ran into our 5-minute timeout. The narrow inter-quartile ranges in Figure 9.3 show that the running time for HT and SP[d] clearly depends on the number of crossings between cluster boundaries and edges in the given instance, while it is much more scattered for ILP. Still, all instances with less than 20 such crossings could be solved by ILP. For HT, we can see that the verification and embedding of yes-instances has an overhead of at least an order of magnitude over the non-verifying HT-f. The running times for HT on no-instances as well as the times for HT-f on any type of instance are the same, showing that the overhead is solely caused by the verification while the base running time is always the same. For the larger instances in this test set, SP[d] is an order of magnitude faster than HT-f. For SP[d], we also see a division between yes- and no-instances, where the latter can be solved faster, but also with more scattered running times. This is probably due to the fact that the test can fail at any (potentially very early) reduction step or when solving the reduced instance. Furthermore, we additionally generate an embedding for positive instances, which may cause the gap between yes- and no-instances.

The running times on dataset C-NCP are shown in Figure 9.4. The result counts in Table 9.2 show that only a small fraction of the instances are positive. With only up to 300 cluster-edge crossings these instances are also comparatively small. The growth of the running times is similar to the one already observed for the smaller instances in Figure 9.3. HT-f now runs into the timeout for almost all yes-instances of size 200 or larger, and both HT and HT-f time out for all instances of size 1500 and larger. The ILP only manages to solve very few of the instances,

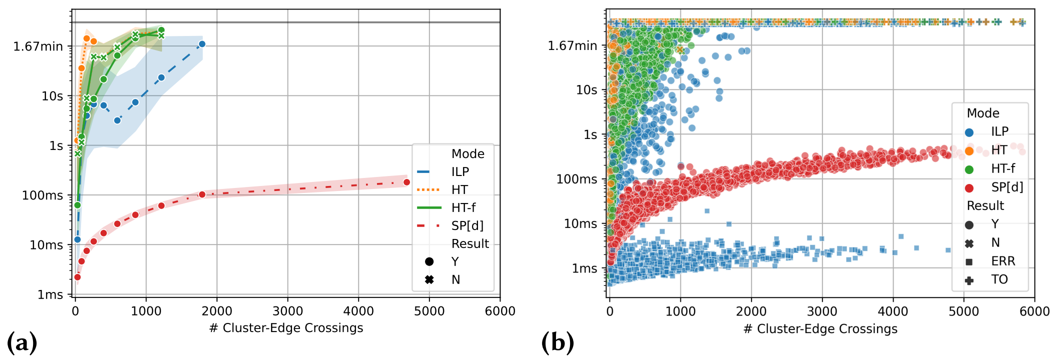


Figure 9.5: Median running times **(a)** and scatter plot **(b)** on dataset C-MED.

often reporting an "undefined optimization result for c-planarity computation" as error; see Table 9.2. The algorithms all agree on the result if they do not run into a timeout or abort with an error, except for one instance that HT classifies as negative while SP[d] found a (positive) solution and also verified its correctness using the generated embedding as certificate. This is even though the Hanani-Tutte approach by Gutwenger et al. [GMS14] should answer "no" only if the instance truly is negative. Figure 9.2 (b) shows a minimal minor of the instance for which the results still disagree.

The running times on dataset C-MED with only positive instances shown in Figure 9.4 are in accordance with the previous results. We now also see more false-negative answers from the HT approach, which points to an error in its implementation; see also Table 9.2. The plots clearly show that our approach is much faster than all others. As the Synchronized Planarity reduction fails at an arbitrary step for negative instances, the running times of positive instances form an upper bound for those of negative instances. As we also see verifying positive instances to obtain an embedding as far more common use-case, we focus our following engineering on this case.

9.3 Engineering Synchronized Planarity

In this section, we study how degrees of freedom in the Synchronized Planarity algorithm can be used to improve the running times on yes-instances. The algorithm makes little restriction on the order in which pipes are processed, which gives great freedom to the implementation for choosing the pipe it should process next. In Section 9.3.1 we investigate the effects of deliberately choosing the next pipe depending on its degree and whether removing it requires generation of an

embedding tree. As mentioned by the original description of the Synchronized Planarity algorithm, there are two further degrees of freedom in the algorithm, both concerning pipes where both endpoints are block-vertices. The first one is that if both endpoints additionally lie in different connected components, we may apply either PropagatePQ or (EncapsulateAnd)Join to remove the pipe. Joining the pipe directly removes it entirely instead of splitting it into multiple smaller ones, although at the cost of generating larger connected components. The second one is for which endpoint of the pipe to compute an embedding tree when applying PropagatePQ. Instead of computing only one embedding tree, we may also compute both at once and then use their intersection. This preempts multiple following operations propagating back embedding information individually for each newly-created smaller pipe. We investigate the effect of these two decisions in Section 9.3.2. Lastly, we investigate an alternative method for computing embedding trees in Section 9.3.3, where we employ a more time-consuming algorithm that in return yields embedding trees for all vertices of a biconnected component simultaneously instead of just for a single vertex.

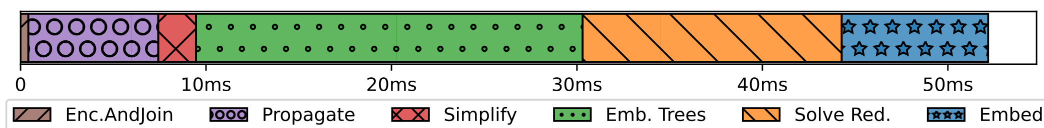


Figure 9.6: Average time spent on different operations for SP[d] on C-MED.

To gain an initial overview over which parts could benefit the most from improvements, Figure 9.6 shows how the running time is distributed across different operations, averaged over all instances in C-MED. It shows that with more than 20ms, that is roughly 40 % of the overall running time, a large fraction of time is spent on generating embedding trees, while the actual operations contribute only a minor part of roughly 18 % of the overall running time. 27 % of time is spent on solving and embedding the reduced instance and 15 % is spent on undoing changes to obtain an embedding for the input graph. Thus, the biggest gains can probably be made by reducing the time spent on generating embedding information in the form of embedding trees. We use this as rough guideline in our engineering process.

Dataset Generation

To tune the running time of our algorithm on larger instances, we increased the size of the generated instances by a factor of 100 by changing the parameters of our own cluster-planar instance generator to $n \in \{100, 500, 1000, 5000, 10\,000,$

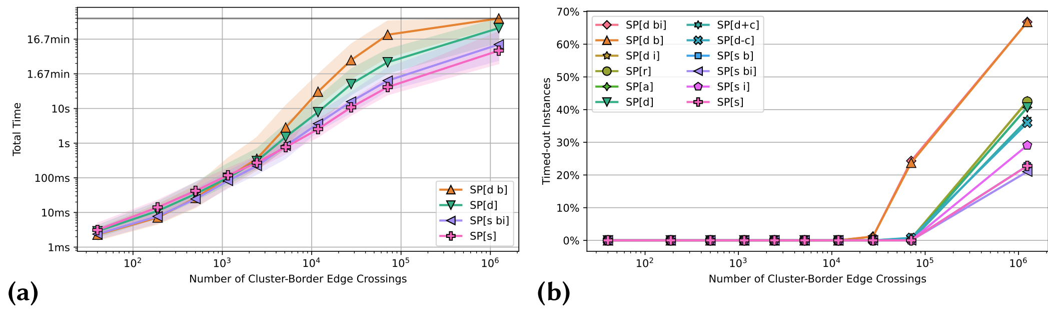


Figure 9.7: C-LRG median absolute running times **(a)** and fraction of timeouts **(b)**. Each marker again corresponds to a bin containing 10 % of the instances.

50 000, 100 000}, $d \in \{1, 10, 100\}$, $c \in \{3, 5, 10, 25, 50, 100, 1000\}$ for dataset C-LRG. This yields 6615 instances, out of which 5096 are non-trivial after preprocessing; see Table 9.1.

In addition to the Clustered Planarity dataset we also generate a dataset that uses the reduction from Connected SEFE-2. We do so by generating a random connected and planar embedded graph as shared graph. Each exclusive graph contains further edges which are obtained by randomly splitting the faces of the embedded shared graph until we reach a desired density. For the shared graphs, we generate three instances for each combination of $n \in \{100, 500, 1000, 2500, 5000, 7500, 10\,000\}$ nodes and $m \in \{n, 1.5n, 2n, 2.5n\}$ edges. For $d \in \{0.25, 0.5, 0.75, 1\}$, we then add $(3n - 6 - m) \cdot d$ edges to each exclusive graph, i.e., the fraction d of the number of edges that can be added until the graph is maximal planar. We also repeat this process three times with different initial random states for each pair of shared graph and parameter d. This leads to the dataset SEFE-LRG containing 1008 instances.

We also generate a dataset of Synchronized Planarity instances by taking a random planar embedded graph and adding pipes between vertices of the same degree, using a bijection that matches their current rotation. The underlying graphs are comprised of three instances for each combination of $n \in \{100, 500, 1000, 5000, 10\,000, 50\,000, 100\,000\}$ nodes, $m \in \{1.5n, 2n, 2.5n\}$ edges, and $d \in \{1, 10, 100\}$ distinct connected components. Note that we do not include graphs that would have no edges, e.g., those with $n = 100$ and $d = 100$. For each input graph, we generate three random Synchronized Planarity instances with $p \in \{0.05n, 0.1n, 0.2n\}$ pipes. This leads to the dataset SP-LRG containing 1587 instances.

Altogether, our six datasets contain 28 339 instances in total. For the test runs on these large instances, we increase the timeout to 1 hour. Figure 9.7 (a) shows

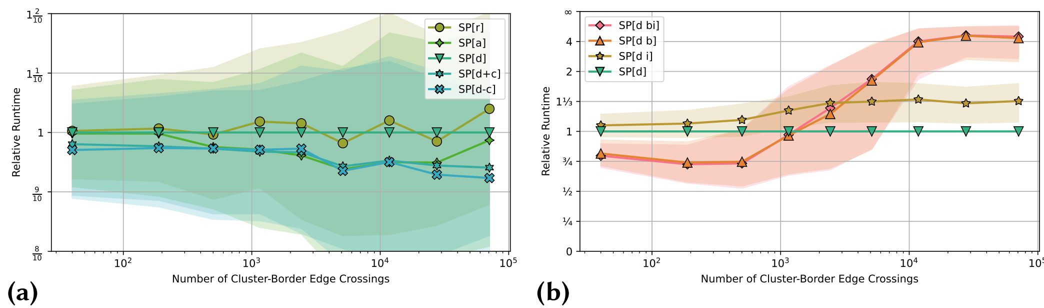


Figure 9.8: Relative running times when **(a)** sorting by pipe degree or applicable operation and **(b)** when handling pipes between block-vertices via intersection or join. Note the different scales on the y-axis.

the result of running our baseline variant SP[d] of the Synchronized Planarity algorithm (together with selected further variants of the algorithm from subsequent sections) on dataset C-LRG. Note that, because the dataset spans a wide range of instance sizes and thus the running times also span a range of different magnitudes, the plot uses a log scale for both axes. Figure 9.7 (b) shows the fraction of runs that timed out for each variant.

9.3.1 Pipe Ordering

To be able to deliberately choose the next pipe, we keep a heap of all pipes in the current instance, where the ordering function can be configured. Note that the topmost pipe from this heap may not be feasible, in which case we will give priority to the close-by pipe of higher degree that blocks the current pipe from being feasible; see Lemma 6.7 for how to find this pipe. We compare the baseline variant SP[d] sorting by descending (i.e. largest first) degree with the variant SP[a] sorting by ascending degree, and SP[r] using a random order. Note that for these variants, the ordering does not depend on which operation is applicable to a pipe or whether this operation requires the generation of an embedding tree. To see whether making this distinction affects the running time, we also compare the variants SP[d+c], which prefers to process pipes on which EncapsulateAndJoin can be applied, and SP[d-c], which defers such pipes to the very end, processing pipes requiring the generation of embedding trees first.

To make the variants easier to compare, Figure 9.8 (a) shows running times relative to that of the baseline SP[d]. Note that we do not show the median of the last bin, in which up to 70 % of the runs timed out, while this number is far lower for all previous bins; see Figure 9.7 (b). Figure 9.8 (a) shows that the median running

times differ by less than 10 % between these variants. The running time of SP[r] seems to randomly alternate between being slightly slower and slightly faster than SP[d]. SP[d] is slightly slower than SP[a] for all bins except the very first and very last, indicating a slight advantage of processing small pipes before bigger ones on these instances. Interestingly, SP[d] is also slower than both SP[d+c] and SP[d-c] for all bins. The fact that these two variants have the same speed-ups indicates that EncapsulateAndJoin should not be interleaved with the other operations, while it does not matter whether it is handled first or last. Still, the variance in relative running times is high and none of the variants is consistently faster on a larger part of the instances (see Section 9.4.2 for a more in-depth analysis of this). To summarize, the plots show a slight advantage for not interleaving operation EncapsulateAndJoin with the others or sorting by ascending degree, but this advantage is not significant in the statistical sense; see Section 9.4.2. We keep SP[d] as the baseline for our further analysis.

9.3.2 Pipes with two Block-Vertex Endpoints

Our baseline always processes pipes where both endpoints are block-vertices by applying PropagatePQ or SimplifyMatching based on the embedding tree of an arbitrary endpoint of the pipe. Alternatively, if the endpoints lie in different connected components, such pipes can also be joined directly by identifying their incident edges as in the second step of EncapsulateAndJoin. This directly removes the pipe entirely instead of splitting into further smaller pipes, although it also results in larger connected components. We enable this joining in variant SP[d b]. As a second alternative, we may also compute the embedding trees of both block-vertices and then propagate their intersection. This preempts the multiple following operations propagating back embedding information individually for each newly-created smaller pipe. We enable this intersection in variant SP[d i]. Variant SP[d bi] combines both variants, preferring the join and only intersecting if the endpoints are in the same connected component. We compare the effect of differently handling pipes with two block-vertex endpoints in variants SP[d b], SP[d i] and SP[d bi] with the baseline SP[d], which computes the embedding tree for an arbitrary endpoint and only joins pipes where both pipes are cut-vertices.

Figure 9.8 (b) shows that SP[d b] (and similarly SP[d bi]) is faster by close to 25 % on instances with less than 1000 cluster-border edge crossings, but quickly grows 5 times slower than SP[d] for larger instances. This effect is also visible in the absolute values of Figure 9.7 (a). This is probably caused by the larger connected

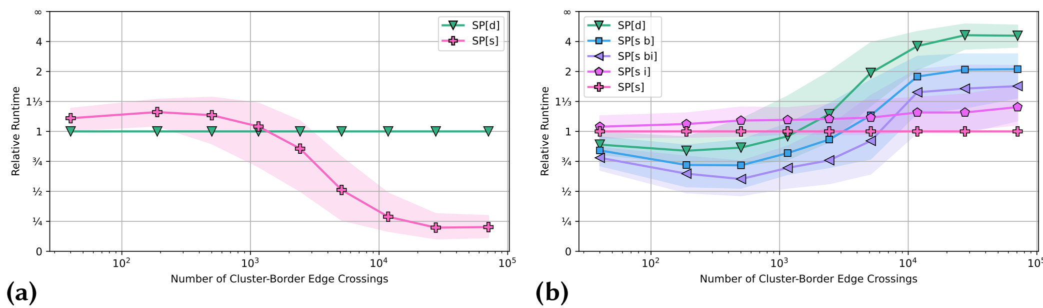


Figure 9.9: Relative running times for **(a)** SPQR-tree batched embedding tree generation and **(b)** for different variants thereof.

components (see the last column of Table 9.3), which make the computation of embedding trees more expensive. Only inserting an embedding tree instead of the whole connected component makes the embedding information of the component directly available in a compressed form without the need to later process the component in its entirety again. Figure 9.8 (b) also shows that SP[d i] is up to a third slower than SP[d], indicating that computing both embedding trees poses a significant overhead while not yielding sufficiently more information to make progress faster. We also evaluated combinations of the variants from this section with the different orderings from the previous section, but observed no notable differences in running time behavior. The effects of the variants from this section always greatly outweigh the effects from the different orderings. To summarize, as the plots only show an advantage of differently handling pipes between block-vertices for small instances, but some strong disadvantages especially for larger instances, we keep SP[d] as our baseline.

9.3.3 Batched Embedding Tree Generation

Our preliminary analysis showed that the computation of embedding trees consumes a large fraction of the running time (see Figure 9.6), which cannot be reduced significantly by using the degrees of freedom of the algorithm studied in the previous two sections. To remedy the overhead of recomputing embedding trees multiple times we now change the algorithm to no longer process pipes one-by-one, but to process all pipes of a biconnected component in one batch. This is facilitated by an alternative approach for generating embedding trees not only for a single vertex, but for all vertices of a biconnected component. The embedding tree of a vertex v can be derived from the SPQR-tree using the approach described by Bläsius et al. [BR16b, Section 2.5] (see also Section 7.4.1): Each occurrence of v in a “parallel”

skeleton of the SPQR-tree corresponds to a (PC-tree) P-node in the embedding tree of v, each occurrence in a "rigid" to a (PC-tree) C-node. This derivation can be done comparatively quickly, in time linear in the degree of v. Thus, once we have the SPQR-tree of a biconnected component available, we can apply all currently feasible `PropagatePQ` and `SimplifyMatching` operations in a single batch with little overhead. The SPQR-tree computation takes time linear in the size of the biconnected component, albeit with a larger linear factor than for the linear-time planarity test that yields only a single embedding tree. In a direct comparison with the planarity test, this makes the SPQR-tree the more time-consuming approach.

We enable the batched embedding tree computation based on SPQR-trees in variant SP[s]. Figures 9.7 (a) and 9.9 (a) show that for small instances, this yields a slowdown of close to a third. Showing a behavior inverse to SP[d b], SP[s] grows faster for larger instances and its speed-up even increases to up to 4 times as fast as the baseline SP[d]. This makes SP[s] the clear champion of all variants considered so far. We will thus use it as baseline for our further evaluation, where we combine SP[s] with other, previously considered flags.

9.3.4 SPQR-Batch Variations

Figure 9.9 (b) switches the baseline between the two variants shown in Figure 9.9 (a) and additionally contains combinations of the variants from Section 9.3.2 with the SPQR-batch computation. As in Figure 9.8 (b), the intersection of embedding trees in SP[s i] is consistently slower, albeit with a slightly smaller margin. The joining of blocks in SP[s b] also shows a similar behavior as before, starting out 25 % faster for small instances and growing up to 100 % slower for larger instances. Again, this is probably because too large connected components negatively affect the computation of SPQR-trees. Still, the median of SP[s b] is consistently faster than SP[d]. Different to before, SP[s bi] is now faster than SP[s b], making it the best variant for instances with up to 5000 cluster-border edge crossings. This is probably because in the batched mode, there is no relevant overhead for obtaining a second embedding tree, while the intersection does preempt some following operations. To summarize, for instances up to size 5000, SP[s bi] is the fastest variant, which is outperformed by SP[s] on larger instances. This can also be seen in the absolute running times in Figure 9.7 (a), where SP[s] is more than an order of magnitude faster than SP[d b] on large instances.

9.4 Further Analysis

In this section, we provide further in-depth analysis of the different variants from the previous section and also analyze their performance on the remaining datasets to give a conclusive judgement. To gain more insights into the runtime behavior, we measured the time each individual step of the algorithm takes when using the different variants. An in-depth analysis of this data is given in Section 9.4.1, where Figure 9.10 also gives a more detailed visualization of per-step timings. The per-step data corroborates that the main improvement of faster variants is greatly reducing the time spent on the generation of embedding trees, at the cost of slightly increased time spent on the solve and embed phases.

To further verify our ranking of variants' running times from the previous sections, we also used a statistical test to check whether one variant is significantly faster than another. The results presented in Section 9.4.2 corroborate our previous results, showing that pipe ordering has no significant effect while the too large connected components and batched processing of pipes using SPQR-trees significantly change the running time.

The results of the remaining datasets SEFE-LRG and SP-LRG are presented in Section 9.4.3 and mostly agree with the results on C-LRG, with SP[d b] clearly being the slowest and SP[s] being the fastest on large instances. The main difference is the magnitude of the overhead generated by large connected components for variants with flag [b].

9.4.1 Detailed Runtime Profiling

Table 9.3 shows the per-step running time information aggregated for variants studied in the previous section. Figure 9.10 in greater detail shows how the running time spent is split on average across the different steps of the algorithm (Figure 9.10 (a)) and then also further drills down on the composition of the individual steps that make the instance reduced (Figure 9.10 (b)), solve the reduced instance (Figure 9.10 (d)), and then derive a solution and an embedding for the input instance by undoing all changes while maintaining the embedding (Figure 9.10 (e)). For variants that use the SPQR-tree for embedding information generation, we also analyze the time spent on the steps of this batch operation (Figure 9.10 (c)). Note that we do not have these measurements available for runs that timed out. To ensure that the bar heights still correspond to the actual overall running times in the topmost plot, we add a bar corresponding to the time consumed by timed-out runs on top. This way, ordering the bars by height yields roughly the same order of variants as we already observed in Figure 9.7 (a).

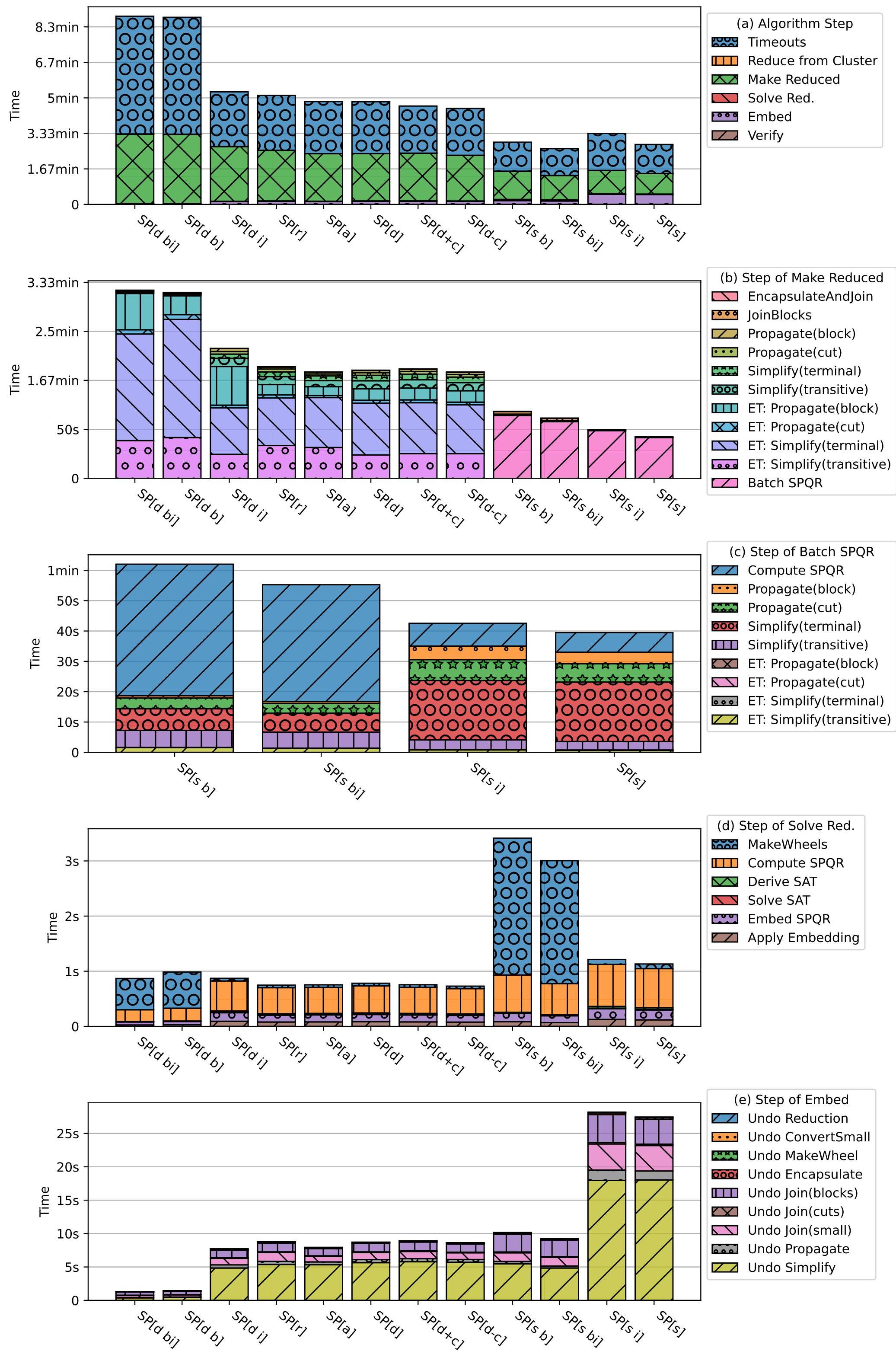


Figure 9.10: The average running time of our different Synchronized Planarity variants.

Mode	Total Time	Make Reduced	Solve Reduced	Embed	Enc.And-Join	Propagate	Simplify	Compute Emb. Tree	Undo Simplify	#Simplify Operations	Max. Bicon. Size
SP[d]	142.68	133.08	0.82	8.78	0.25	5.00	13.79	91.34	5.64	1811	2780
SP[d b]	197.17	194.72	0.99	1.46	0.63	1.36	1.53	186.18	0.42	652	13 021
SP[s]	86.57	57.75	1.25	27.56	0.57	9.84	22.38	7.61	18.03	2696	2890
SP[s b]	93.07	79.25	3.55	10.26	2.92	4.29	12.74	46.31	5.46	1421	22 965
SP[s bi]	81.32	68.90	3.09	9.32	2.51	3.79	11.52	41.16	4.84	1448	23 284

Table 9.3: Average values for different variants of SP on dataset C-LRG. All values, except for the counts in the last two columns, are running times in seconds. The first data column shows the average total running time, followed by how this is split across the three phases. The following four columns show the composition of the running time of the "Make Reduced" step. The last three columns detail information about the "Undo Simplify" step in the "Embed" phase, and the maximum size of biconnected components in the reduced instance.

Figure 9.10 (b) clearly shows that the majority of time during the reduce step is spent on generating embedding information, either in the form of directly computing embedding trees (bars prefixed with "ET") or by computing SPQR trees. This can also be seen by comparing column "Make Reduced" in Table 9.3 with column "Compute Emb Tree". Only for the fastest variants, those with flag [s] and without [b], the execution of the actual operations of the algorithm becomes more prominent over the generation of embedding information in Figure 9.10 (c). Here, the terminal case of the SimplifyMatching operation (described in the bottom left part of Figure 7.6) now takes the biggest fraction of time, and actually also a bigger absolute amount of time than for the other, slower variants with flag [b] enabled. This is probably because, instead of being joined as with flag [b] enabled, here pipes between block-vertices are split by PropagatePQ into multiple smaller pipes, which then need to be removed by SimplifyMatching. This leads to the variants without [b] needing, on average, roughly two to three times as many SimplifyMatching applications as those with [b]; see Table 9.3.

The larger biconnected components caused by [b] may also be the reason why the insertion of wheels takes a larger amount of time for variants with [b] in the solving phase shown in Figure 9.10 (d). When replacing a cut-vertex by a wheel, all incident biconnected components with at least two edges incident to the cut-vertex get merged. Updating the information stored with the vertices of the biconnected components is probably consuming the most time here, as undoing the changes by

contracting the wheels is again very fast. Other than the "MakeWheels" part, most time during the solving phase is spent on computing SPQR trees, although both is negligible in comparison to the overall running time.

The running times of the embedding phase given in Figure 9.10 (e) show an interesting behavior as they increase when the "Make Reduced" phase running time decreases, indicating a potential trade-off to be made; see also the "Embed" column in Table 9.3. As the maximum time spent on the "Make Reduced" phase is still slightly larger, variants where this phase is faster while the embedding phase is slower are still overall the fastest. The biggest contribution of running time in the latter phase is the undoing of `SimplifyMatching` operations, which means copying the embedding of one endpoint of a removed pipe to the other. The time spent here roughly correlates with the time spent on applying the `SimplifyMatching` operations in the first place (see Table 9.3).

To summarize, the per-step data corroborates that the main improvement of faster variants is greatly reducing the time spent on the generation of embedding trees, at the cost of slightly increased time spent on the solve and embed phases. Flags [`s`] and [`b`] have the biggest impact on running times, while flag [`i`] and the processing order of pipes do not seem to have a significant influence on the overall running time. While the variants with [`s`] clearly have the fastest overall running times, there is some trade-off between the amounts of time spent on different phases of the algorithm when toggling the flag [`b`].

9.4.2 Statistical Significance

To test whether one variant is (in the statistical sense) significantly faster than another, we use the methodology proposed by Radermacher [Rad20, Section 3.2] for comparing the performance of graph algorithms. For a given graph G and two variants of the algorithm described by their respective running times $f_A(G), f_B(G)$ on G, we want to know whether we have a likelihood at least p that the one variant is faster than the other by at least a factor Δ. To do so, we use the binomial sign test with advantages as used by Radermacher [Rad20], where we fix two values $p \in [0, 1]$ and $\Delta \geq 1$, and study the following hypothesis given a random graph G from our dataset: Inequality $f_A(G) \cdot \Delta < f_B(G)$ holds with probability π, which is at least p. The respective null hypothesis is that the inequality holds with probability less than p. Note that this is an experiment with exactly two outcomes (the inequality holding or not), which we can independently repeat on a sequence of n graphs and obtain the number of instances k for which the inequality holds. Using the binomial test, we can check the likelihood of obtaining at most k successes by drawing n times from a binomial distribution with probability p. If this likelihood is below a

given significance level $\alpha \in [0, 1]$, that is the obtained result is unlikely under the null hypothesis, we can reject the null hypothesis that the inequality only holds with a probability less than p.

Fixing the significance level to the commonly-used value $\alpha = 0.05$, we still need to fix values for p and Δ to apply this methodology in practice. We will use three different values for $p \in [0.25, 0.5, 0.75]$, corresponding to the advantage on a quarter, half, and three quarters of the dataset. To obtain values for Δ, we will split our datasets evenly into two halves $\mathcal{G}_{\text{train}}$ and $\mathcal{G}_{\text{verify}}$, using the $\mathcal{G}_{\text{train}}$ to obtain an estimate for Δ and $\mathcal{G}_{\text{verify}}$ to verify this value. For a given value of p, we set Δ' to the largest value such that $f_A(G) \cdot \Delta' < f_B(G)$ holds for $p \cdot |\mathcal{G}_{\text{train}}|$ instances. To increase the likelihood that we can reject the null hypothesis in the verification step on $\mathcal{G}_{\text{verify}}$, we will slightly discount the obtained value of Δ', using $\Delta = \min(1, c \cdot \Delta')$ instead with c set to 0.75.

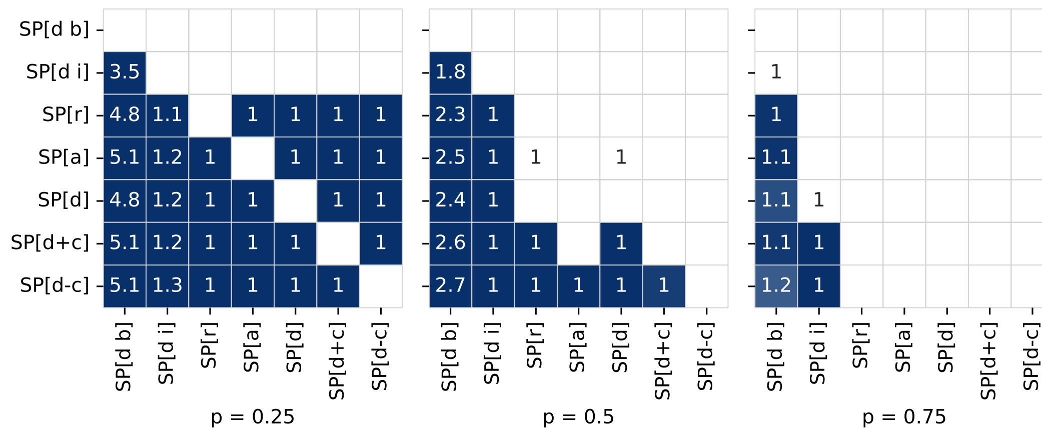


Figure 9.11: Advantages of variants without flag [s] on C-LRG instances of size at least 5000. Blue cell backgrounds indicate significant values, while in cells with white background, we were not able to reject the null-hypothesis with significance $\alpha = 0.05$. Empty cells indicate that the fraction where the one algorithm is better than the other is smaller than p.

Applying this methodology, Figure 9.11 compares the pairwise advantages of the variants from Sections 9.3.1 and 9.3.2. We see that SP[d i] and especially SP[d b] are significantly slower than the other variants: for the quarter of the dataset with the most extreme differences, the advantage rises up to a 5-fold speed-up for other variants, while slight advantages still persist when considering three quarters of instances. Conversely, not even on a quarter of instances are SP[d i] and SP[d b] faster than other variants. Comparing the remaining variants with each other, we see that each variant has at least a quarter of instances where it is slightly faster than the other variants, but always with no noticeable advantage, that is $\Delta = 1$. This is not surprising as the relative running times are scattered evenly above and below the baseline in Figure 9.8 (a). For half of the dataset, SP[d-c] is still slightly

faster than other variants, while no variant from Section 9.3.1 is faster than another for at least three quarters of instances. To summarize, our results here corroborate the findings from Sections 9.3.1 and 9.3.2, with SP[d i] and SP[d b] as the clearly slowest variants. While there is no clear winner among the other variants, at least SP[d-c] is slightly faster than the others on half of the dataset, but still has no noticeable advantage.

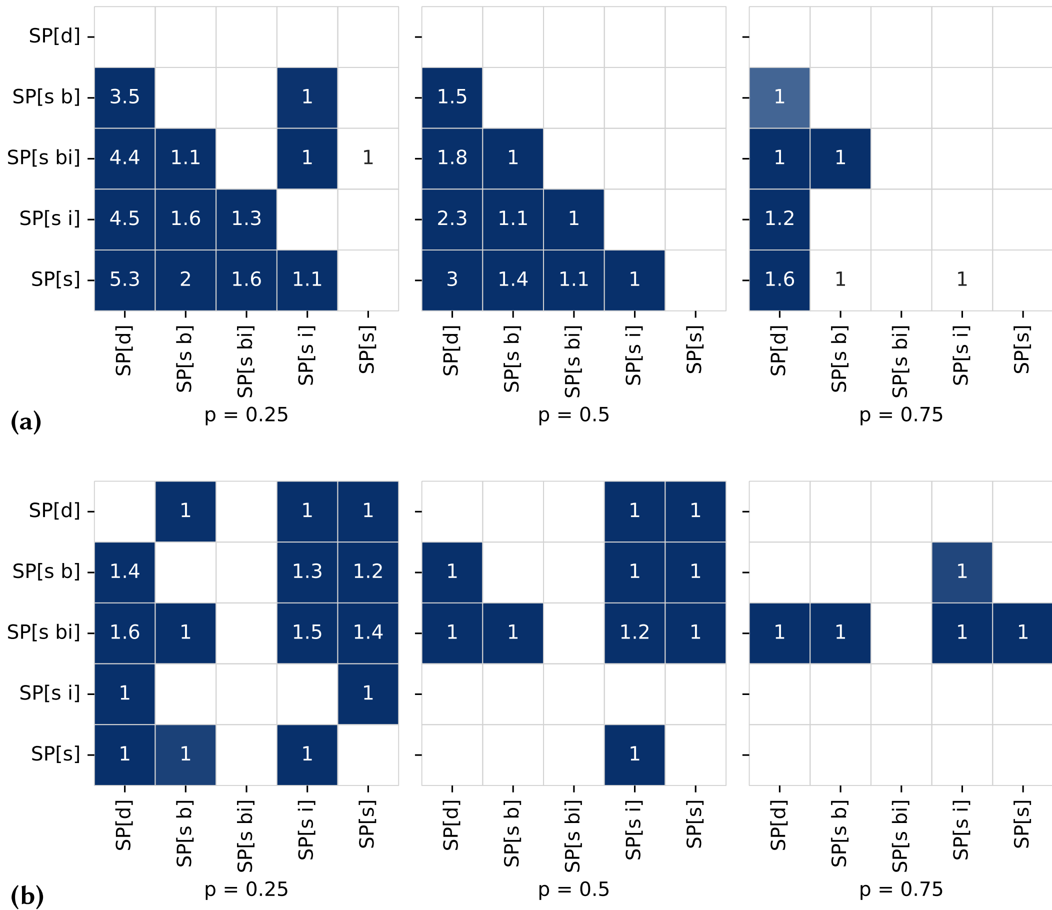


Figure 9.12: Advantages of variants with flag [s] on C-LRG instances of size at least 5000 **(a)** and at most 5000 **(b)**.

Figures 9.12 (a) and 9.12 (b) compare the pairwise advantages of the variants from Sections 9.3.3 and 9.3.4 (see also Figure 9.9 (b)) for instances with more and less than 5000 cluster-border edge crossings, respectively. For the larger instances of Figure 9.12 (a), the variants with flag [s] outperform SP[d] on at least 75 % of instances, with advantages as high as a factor of 5 on at least a quarter of instances. Furthermore, SP[s] outperforms the variants with additional flags [b] and [i] on

at least half of all instances. Considering 75 % of all instances, the only significant result is that SP[s bi] outperforms SP[s b] but with no advantage, i.e. $\Delta = 1$. For the smaller instances of Figure 9.12 (b), the comparison looks vastly different. Here, SP[s bi] outperforms all other variants on at least 75 % of instances, although its advantage is not large, with only up to 1.6 even on the most extreme quarter of the dataset. Furthermore, variants SP[d] and SP[s b] outperform variants SP[s i] and SP[s] on half of the dataset, but again with no noticeable advantage, that is $\Delta = 1$. To summarize, our results are again in accordance with those from Sections 9.3.3 and 9.3.4, where for large instances variant SP[s] is the fastest, whereas for smaller instances SP[s bi] is superior.

9.4.3 Other Problem Instances

Running the same evaluation on the datasets SEFE-LRG and SP-LRG yielded absolute running times with roughly the same orders of magnitude as for C-LRG, see the left plots in Figures 9.13 to 9.15 (but note that the plots show different ranges on the x-axis while having the same scale on the y-axis). The right plots in the figures again detail the running times relative to SP[d]. For SP-LRG, the relative behavior is similar to the one observed on C-LRG. The two major differences concern variants with flag [b]. Variant SP[d b(i)] is not faster than SP[d] on small instances and also sooner grows slower on large instances. Similarly, SP[s b(i)] is not much faster than SP[d] on small instances, and its speed-up over SP[d] for larger instances has a dent where it returns to having roughly the same speed as SP[d] around size 1000. On a large scale, this behavior indicates that the slowdown caused by large connected components is even worse in dataset SP-LRG. For SEFE-LRG, the instances are less evenly distributed in terms of their total pipe degree, as the total pipe degree directly corresponds to the vertex degrees in the SEFE-2 instance. Regarding the relative running time behavior, we still see that SP[d bi] is much slower and SP[s (i)] much faster than SP[d]. For the remaining variants, the difference to SP[d] is much smaller than in the two other datasets. This indicates that the size of connected components does not play an as important role in this dataset as before.

9.5 Conclusion

In this chapter, we described the first practical implementation of Synchronized Planarity, which generalizes many constrained planarity problems such as Clustered Planarity and Connected SEFE-2. We evaluated it on more than 28 000 instances stemming from different problems. Using the quadratic algorithm from Chapter 6, instances with 100 vertices are solved in milliseconds, while we can still

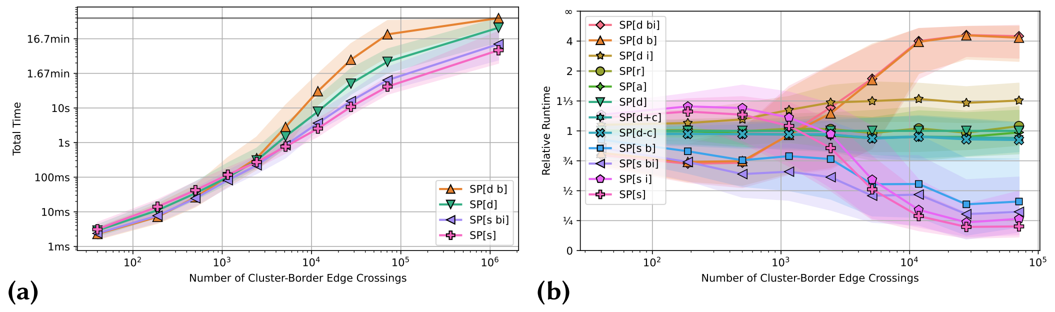


Figure 9.13: Absolute **(a)** and relative **(b)** running times with regard to SP[d] for C-LRG.

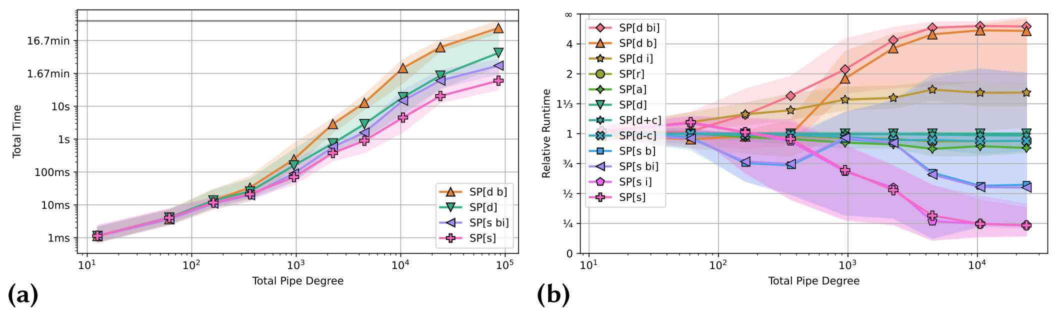


Figure 9.14: Absolute **(a)** and relative **(b)** running times with regard to SP[d] for SP-LRG.

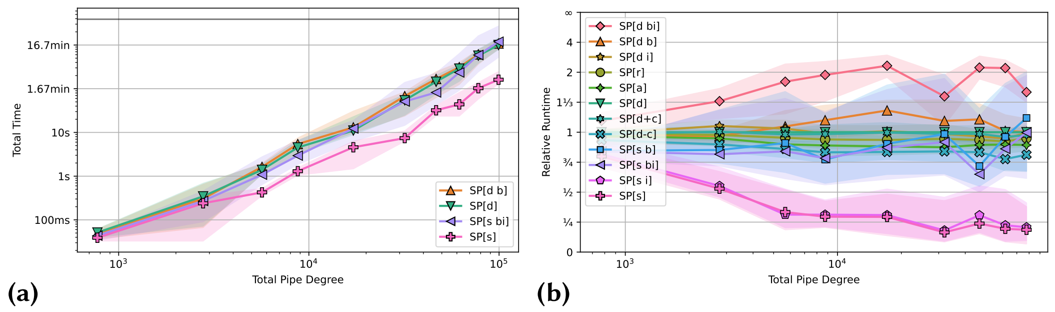


Figure 9.15: Absolute **(a)** and relative **(b)** running times with regard to SP[d] for SEFE-LRG.

solve most instances with up to 100 000 vertices within minutes. This makes our implementation at least an order of magnitude faster than all other Clustered Planarity implementations, which corroborates its theoretical guarantees in practice. Analyzing our running times in more detail, we find the generation of embedding information in the form of embedding trees to be by far the most time-consuming, while the actual operations of the algorithm that reduce and solve the instance are comparatively fast. We apply algorithm engineering and use the various degrees of freedom of the algorithm to speed up computation times by up to an order of magnitude. The main result here is that the batched computation of embedding information we devise using SPQR-trees produces a major speed-up. Tuning some other variables produces a speed-up only in parts of the algorithm while slowing down others, which shows that further speed-ups may be more challenging to achieve and that trade-offs may have to be made. One possible approach could be implementing the dynamically-maintained SPQR-tree described in Chapter 7, which also yields a further theoretical speed-up. As contribution towards future work in the field of graph drawing, we also see that our implementation can be used as reference for the implementation of more specialized, but potentially faster constrained planarity algorithms, which proved challenging in the past [Brü21].

10 Conclusions

In this thesis, we studied theoretical as well as practical aspects of constrained planarity. The theoretical Part I introduced two new algorithms for solving the Partially Embedded and Synchronized Planarity problems and, through given reductions, many other constrained variants planarity variants they generalize. In the practical Part II, we presented and engineered our implementation of the PC-tree as data structure that both algorithms rely upon and, based on this, also the implementation of our quadratic Synchronized Planarity solution. While the individual chapters close with their individual conclusions and topic-specific outlooks, we here want to give an overarching summary with a superordinate perspective on the work. At the very beginning of the thesis, we started with an updated overview over the most important variants of constrained planarity and their relationship in Chapter 3, also highlighting the contributions in this thesis. We publish the contents of this chapter as a website at constrained-planarity.github.io, with the possibility to collaboratively keep the information updated with the results of future research. In Chapter 4, we gave an extensive explanation of the basic algorithm for standard (i.e., unconstrained) planarity and the PC-tree data structure it relies upon.

On a high level, this PC-tree data structure is a central concept that pertains to all further parts of this thesis. In Chapter 5, the PC-tree forms the central part of our linear-time solution for the problem Partially Embedded Planarity based on the vertex-addition planarity test of Haeupler and Tarjan [HT08]. Augmenting the PC-tree to also encode the further constraints stemming from Partially Embedded Planarity allows us to solve this problem with an only slightly-adapted variant of the standard planarity test. In Chapter 6, we gave a quadratic-time algorithm for resolving the synchronization of vertex rotations modeled by Synchronized Planarity. Here, being able to succinctly describe and "communicate" all restrictions a component has on the rotations of a vertex in the form of an embedding PC-tree is a key ingredient to the efficiency of our algorithm. For both solutions, this reliance on local orders described by PC-trees contrasts previous algorithms, which mainly focus on the global perspective on embedding possibilities in the form of the SPQR-tree [ADP09; Ang+12; Ang+15b; Cor+08; HN09]. Instead, our solutions step-by-step process local embedding choices in the form of PC-trees to obtain simple and efficient algorithms.

While we focused on the dynamic maintenance of SPQR-trees in Chapter 7, we also directly applied this result to more efficiently obtain the embedding PC-trees central to our algorithm from Chapter 6. Without having to make changes to the Synchronized Planarity algorithm itself, this allowed us to improve its asymptotic running time even further. Our experimental evaluation of this algorithm in Chapter 9 showed that also in practice, the main effort is not applying the different operations we describe, but generating the embedding PC-trees they rely on. To be able do so in the first place, we needed to develop an appropriate implementation of the PC-tree data structure, which we described in Chapter 8. Here, we showed that, among different available alternatives, the choice of the most efficient in-memory representation as well as better implementation of operations has great effects on the practical performance of the data structure. These improvements directly pertain to the algorithms that rely on the data structure, which we showed on the example of planarity test implementations. In this thesis, the PC-tree data structure thus spans theory, from the conceptual modelling and through allowing efficient algorithms, to practice, where it is key to good performance.

Interestingly, we also found some gaps between the possibilities of theoretical and practical considerations. While using Union-Find in our second PC-tree implementation entails an asymptotic slow-down in theory, this lead to a factor two speed-up in practice. Furthermore, while our analysis of the running time of Synchronized Planarity already predicted the generation of embedding trees to be a major effort,[18] it is also indifferent to the different ways in which we can obtain these embedding trees in practice. Counterintuitively, using a method with a higher constant factor in its linear running time here leads to a better practical running time. In this case, this is because computing global embedding choices in the form of an SPQR-tree in a sense parallelizes multiple computations of individual local embedding choices in the form of PC-trees.

To summarize, we add crucial instruments to the algorithmic toolbox for treating – that is especially modeling and solving – constrained planarity problems both in theory and in practice: On the one hand, we provide a linear-time solution to problems that individually constrain the rotations of vertices of the graph. Here, we use adapted PC-trees to represent all planar embedding possibilities of a subgraph growing step by step while respecting given additional constraints. On the other hand, we provide a quadratic-time solution to many more-involved constrained planarity variants by modelling their constraints as synchronization between vertex rotations. Here, we use PC-trees to communicate the restrictions

18 This also includes the problems due to too-big connected components resulting from joining pipes with two block-vertices as endpoints.

of rotations between the two vertices involved in a synchronization constraint. In our solution, this is combined with two further operations that each work to reduce the complexity of individual constraints. As all three operations can be applied in arbitrary order, this yields a simple and efficient algorithm akin to the Goldberg-Tarjan push-relabel algorithm [GT88]. Albeit especially in comparison to our first algorithm, we found this property of the second algorithm to not only be of advantage. The algorithm incrementally modifies the structure of the instance to obtain a solution instead of generating a representation of possible solutions. While the resulting instance clearly does represent a solution, it is hard to see how it correlates with the input instance. Future work may attempt to better understand this relationship between input and solution together with the high-level changes made by the individual operations the algorithm applies. This includes seeking obstructions in the form of a Kuratowski-style characterization of e.g. Clustered Planarity, similar to previous results for Partially Embedded Planarity [JKR13] or the detection and extension of partial representations of interval graphs [KS18; LB62]. We believe that such insights can be crucial to solving further, yet-unsolved constrained planarity variants. Foremost among them is of course the SEFE-2 problem, but this also applies to further problem variants and combinations such as dynamic, partial or simultaneous Clustered- and Synchronized Planarity.

Bibliography

[ABD12] Patrizio Angelini, Marco Di Bartolomeo, and Giuseppe Di Battista. **Implementing a Partitioned 2-Page Book Embedding Testing Algorithm**. In: *Proceedings of the 20th International Symposium on Graph Drawing (GD'12)*. Ed. by Walter Didimo and Maurizio Patrignani. Vol. 7704. Lecture Notes in Computer Science. Springer, 2012, 79–89. DOI: 10.1007/978-3-642-36763-2_8 (see pages 21, 170).

[ABR14] Patrizio Angelini, Thomas Bläsius, and Ignaz Rutter. **Testing Mutual Duality of Planar Graphs**. *International Journal of Computational Geometry & Applications* 24:4 (2014), 325–346. DOI: 10.1142/S0218195914600103 (see page 110).

[AD16] Patrizio Angelini and Giordano Da Lozzo. **SEFE = C-Planarity?** *The Computer Journal* 59:12 (2016), 1831–1838. DOI: 10.1093/comjnl/bxw035 (see pages 21, 60).

[AD19] Patrizio Angelini and Giordano Da Lozzo. **Clustered Planarity with Pipes**. *Algorithmica* 81:6 (2019), 2484–2526. DOI: 10.1007/s00453-018-00541-w (see pages 21, 60).

[ADN15] Patrizio Angelini, Giordano Da Lozzo, and Daniel Neuwirth. **Advancements on SEFE and Partitioned Book Embedding problems**. *Theoretical Computer Science* 575 (2015), 71–89. DOI: 10.1016/j.tcs.2014.11.016 (see pages 21, 98).

[ADP09] Patrizio Angelini, Giuseppe Di Battista, and Maurizio Patrignani. **Finding a Minimum-depth Embedding of a Planar Graph in $O(n^4)$ Time**. *Algorithmica* 60:4 (2009), 890–937. DOI: 10.1007/s00453-009-9380-6 (see page 195).

[AFT19] Hugo A. Akitaya, Radoslav Fulek, and Csaba D. Tóth. **Recognizing Weak Embeddings of Graphs**. *ACM Transactions on Algorithms* 15:4 (2019), 1–27. DOI: 10.1145/3344549 (see pages 59, 60).

[Ang+10] Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Vít Jelínek, Jan Kratochvíl, Maurizio Patrignani, and Ignaz Rutter. **Testing Planarity of Partially Embedded Graphs**. In: *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*. Ed. by Moses Charikar. SIAM, 2010, 202–221. DOI: 10.1137/1.9781611973075.19 (see pages 3, 5, 33).

[Ang+12] Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Maurizio Patrignani, and Ignaz Rutter. **Testing the simultaneous embeddability of two graphs whose intersection is a biconnected or a connected graph**. *Journal of Discrete Algorithms* 14 (2012), 150–172. DOI: 10.1016/j.jda.2011.12.015 (see pages 21, 195).

[Ang+15a] Patrizio Angelini, Giordano Da Lozzo, Giuseppe Di Battista, Fabrizio Frati, and Vincenzo Roselli. **The importance of being proper: (In clustered-level planarity and T-level planarity)**. *Theoretical Computer Science* 571 (2015), 1–9. DOI: 10.1016/j.tcs.2014.12.019 (see page 87).

[Ang+15b] Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Vít Jelínek, Jan Kratochvíl, Maurizio Patrignani, and Ignaz Rutter. **Testing Planarity of Partially Embedded Graphs**. *ACM Transactions on Algorithms* 11:4 (2015), 32:1–32:42. DOI: 10.1145/2629341 (see pages 2, 5, 13, 19, 21, 33–36, 51, 55, 56, 84, 195).

[Ang+16] Patrizio Angelini, Giordano Da Lozzo, Giuseppe Di Battista, and Fabrizio Frati. **Strip Planarity Testing for Embedded Planar Graphs**. *Algorithmica* 77:4 (2016), 1022–1059. DOI: 10.1007/s00453-016-0128-9 (see pages 19, 20, 87, 105).

[APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. **A linear-time algorithm for testing the truth of certain quantified boolean formulas**. *Information Processing Letters* 8:3 (Mar. 1979), 121–123. ISSN: 0020-0190. DOI: 10.1016/0020-0190(79)90002-4 (see page 74).

[ART21] Patrizio Angelini, Ignaz Rutter, and Sandhya T. P. **Extending Partial Orthogonal Drawings**. *Journal of Graph Algorithms and Applications* 25:1 (2021), 581–602. DOI: 10.7155/jgaa.00573 (see page 33).

[Bac04] Christian Bachmaier. **Circle planarity of level graphs**. PhD thesis. University of Passau, Germany, 2004. URL: http://www.opus-bayern.de/uni-passau/volltexte/2004/38/index.html (see page 171).

[BBF05] Christian Bachmaier, Franz-Josef Brandenburg, and Michael Forster. **Radial Level Planarity Testing and Embedding in Linear Time**. *Journal of Graph Algorithms and Applications* 9:1 (2005), 53–97. DOI: 10.7155/jgaa.00100 (see page 19).

[Ben59] S. Benzer. **On the Topology of The Genetic Fine Structure**. *Proceedings of the National Academy of Sciences* 45:11 (1959), 1607–1620. DOI: 10.1073/pnas.45.11.1607 (see page 131).

[BKK97] Therese C. Biedl, Goos Kant, and Michael Kaufmann. **On Triangulating Planar Graphs Under the Four-Connectivity Constraint**. *Algorithmica* 19:4 (1997), 427–446. DOI: 10.1007/PL00009182 (see page 103).

[BKR13] Thomas Bläsius, Stephen G. Kobourov, and Ignaz Rutter. **Simultaneous Embedding of Planar Graphs**. In: *Handbook of Graph Drawing and Visualization*. Ed. by Roberto Tamassia. Chapman and Hall/CRC, 2013. Chap. 11, 349–381. ISBN: 9781584884125. eprint: 1204.5853. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/simultaneous.pdf (see pages 2, 21, 61, 88).

[BKR17] Thomas Bläsius, Annette Karrer, and Ignaz Rutter. **Simultaneous Embedding: Edge Orderings, Relative Positions, Cutvertices**. *Algorithmica* 80:4 (2017), 1214–1277. DOI: 10.1007/s00453-017-0301-9 (see pages 57, 59, 88, 98, 99, 102).

[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 (1976), 335–379. DOI: 10.1016/S0022-0000(76)80045-1 (see pages 1, 5, 10, 13, 23, 31, 34, 123, 131, 134, 149, 151, 164, 173).

[BM89] Daniel Bienstock and Clyde L. Monma. **Optimal enclosing regions in planar graphs**. *Networks* 19:1 (1989), 79–94. DOI: 10.1002/net.3230190107 (see page 103).

[BM90] Daniel Bienstock and Clyde L. Monma. **On the complexity of embedding planar graphs to minimize certain distance measures**. *Algorithmica* 5:1 (1990), 93–109. DOI: 10.1007/bf01840379 (see page 103).

[BM99] John Boyer and Wendy Myrvold. **Stop Minding Your p's and q's: A Simplified $O(n)$ Planar Embedding Algorithm**. In: *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*. SODA '99. Baltimore, Maryland, USA: Society for Industrial and Applied Mathematics, 1999, 140–146. ISBN: 0898714346 (see pages 164, 168).

[Boo75] Kellogg S. Booth. **PQ-tree algorithms**. PhD thesis. University of California, Berkeley, 1975. eprint: https://dl.acm.org/doi/book/10.5555/908019 (see pages 5, 23, 27, 31, 69).

[Boy+04a] John M. Boyer, Pier Francesco Cortese, Maurizio Patrignani, and Giuseppe Di Battista. **Stop Minding Your P's and Q's: Implementing a Fast and Simple DFS-Based Planarity Testing and Embedding Algorithm**. In: *Proceedings of the 11th International Symposium on Graph Drawing (GD'04)*. Springer Berlin Heidelberg, 2004, 25–36. DOI: 10.1007/978-3-540-24595-7_3 (see pages 2, 164, 166, 168).

[Boy+04b] John M. Boyer, Cristina G. Fernandes, Alexandre Noma, and José C. de Pina. **Lempel, Even, and Cederbaum Planarity Method**. In: *Proceedings of the 3rd Workshop on Experimental and Efficient Algorithms (WEA'04)*. Springer Berlin Heidelberg, 2004, 129–144. DOI: 10.1007/978-3-540-24838-5_10 (see page 153).

[BR15] Thomas Bläsius and Ignaz Rutter. **Disconnectivity and relative positions in simultaneous embeddings**. *Computational Geometry. Theory and Applications* 48:6 (2015), 459–478. DOI: 10.1016/j.comgeo.2015.02.002 (see pages 21, 102).

[BR16a] Thomas Bläsius and Ignaz Rutter. **A New Perspective on Clustered Planarity as a Combinatorial Embedding Problem**. *Theoretical Computer Science* 609 (2016), 306–315. DOI: 10.1016/j.tcs.2015.10.011 (see pages 2, 20, 21, 57–59, 80).

[BR16b] Thomas Bläsius and Ignaz Rutter. **Simultaneous PQ-Ordering with Applications to Constrained Embedding Problems**. *ACM Transactions on Algorithms* 12:2 (2016), 16:1–16:46. DOI: 10.1145/2738054 (see pages 20, 59, 61, 70–73, 78, 88–90, 93–95, 105, 123, 131, 183).

[BR17] Guido Brückner and Ignaz Rutter. **Partial and Constrained Level Planarity**. In: *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'17)*. Ed. by Philip N. Klein. SIAM, 2017, 2000–2011. DOI: 10.1137/1.9781611974782.130 (see page 131).

[BR23b] Guido Brückner and Ignaz Rutter. **An SPQR-tree-like embedding representation for level planarity**. en. *Journal of Computational Geometry* (2023), Vol. 14 No. 1 (2023). DOI: 10.20382/JOCG.V14I1A3 (see pages 87, 88).

[Bra+07] Peter Braß, Eowyn Cenek, Christian A. Duncan, Alon Efrat, Cesim Erten, Dan Ismailescu, Stephen G. Kobourov, Anna Lubiw, and Joseph S. B. Mitchell. **On simultaneous planar graph embeddings**. *Computational Geometry. Theory and Applications* 36:2 (2007), 117–130. DOI: 10.1016/j.comgeo.2006.05.006 (see pages 2, 21).

[Bra09] Ulrik Brandes. **The left-right planarity test**. 2009. URL: https://www.uni-konstanz.de/algo/publications/b-lrpt-sub.pdf (see pages 163, 164).

[Brü21] Guido Brückner. **Planarity Variants for Directed Graphs**. PhD thesis. Karlsruhe Institute of Technology, Germany, 2021. URL: https://nbn-resolving.org/urn:nbn:de:101:1-2021080405022988868936 (see pages 1, 2, 131, 168, 171, 193).

[Car17a] Johannes Carmesin. **Embedding Simply Connected 2-Complexes in 3-Space – I. A Kuratowski-type characterisation**. 2017. arXiv: 1709.04642 (see page 60).

[Car17b] Johannes Carmesin. **Embedding Simply Connected 2-Complexes in 3-Space – II. Rotation systems**. 2017. arXiv: 1709.04643 (see page 60).

[Car17c] Johannes Carmesin. **Embedding Simply Connected 2-Complexes in 3-Space – V. A Refined Kuratowski-Type Characterisation**. 2017. arXiv: 1709.04659v3 (see pages 59, 60, 101).

[CFK19] Steven Chaplick, Radoslav Fulek, and Pavel Klavík. **Extending partial representations of circle graphs**. *Journal of Graph Theory* 91:4 (2019), 365–394. DOI: 10.1002/jgt.22436 (see page 33).

[CH17] Alex William Cregten and Hannes Kristján Hannesson. **Implementation of a planarity testing method using PQ-Trees**. Tech. rep. Reykjavík University, 2017. URL: https://skemman.is/bitstream/1946/29618/1/Planarity_testing_with_PQTrees.pdf (see page 154).

[Cha+14] Steven Chaplick, Paul Dorbec, Jan Kratochvíl, Mickaël Montassier, and Juraj Stacho. **Contact Representations of Planar Graphs: Extending a Partial Representation is Hard**. In: *Proceedings of the 40th Workshop on Graph-Theoretic Concepts in Computer Science (WG'14)*. Ed. by Dieter Kratsch and Ioan Todinca. Vol. 8747. Lecture Notes in Computer Science. Springer, 2014, 139–151. DOI: 10.1007/978-3-319-12340-0_12 (see page 33).

[Cha+15] Timothy M. Chan, Fabrizio Frati, Carsten Gutwenger, Anna Lubiw, Petra Mutzel, and Marcus Schaefer. **Drawing Partially Embedded and Simultaneously Planar Graphs**. *Journal of Graph Algorithms and Applications* 19:2 (2015), 681–706. DOI: 10.7155/jgaa.00375 (see page 2).

[Cha+21] Steven Chaplick, Philipp Kindermann, Jonathan Klawitter, Ignaz Rutter, and Alexander Wolff. **Extending Partial Representations of Rectangular Duals with Given Contact Orientations**. In: *Proceedings of the 12th Conference on Algorithms and Complexity (CIAC'21)*. Ed. by Tiziana Calamoneri and Federico Corò. Vol. 12701. Lecture Notes in Computer Science. Springer, 2021, 340–353. DOI: 10.1007/978-3-030-75242-2_24 (see page 33).

[CHH99] Zhi-Zhong Chen, Xin He, and Chun-Hsi Huang. **Finding double Euler trails of planar graphs in linear time [CMOS VLSI circuit design]**. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*. IEEE, 1999. DOI: 10.1109/sffcs.1999.814603 (see page 103).

[Chi+08] Markus Chimani, Carsten Gutwenger, Mathias Jansen, Karsten Klein, and Petra Mutzel. **Computing Maximum C-Planar Subgraphs**. In: *Proceedings of the 16th International Symposium on Graph Drawing (GD'08)*. Ed. by Ioannis G. Tollis and Maurizio Patrignani. Vol. 5417. Lecture Notes in Computer Science. Springer, 2008, 114–120. DOI: 10.1007/978-3-642-00219-9_12 (see pages 170, 173–175).

[Chi+14] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. **The Open Graph Drawing Framework (OGDF)**. In: *Handbook of Graph Drawing and Visualization*. Ed. by Roberto Tamassia. Chapman and Hall/CRC, 2014. Chap. 17, 543–569. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/ogdf.pdf (see pages 8, 132, 136, 164, 173).

[Chi+85] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. **A linear algorithm for embedding planar graphs using PQ-trees**. *Journal of Computer and System Sciences* 30:1 (Feb. 1985), 54–76. DOI: 10.1016/0022-0000(85)90004-2 (see pages 5, 31, 47, 163, 164).

[CK12] Markus Chimani and Karsten Klein. **Shrinking the Search Space for Clustered Planarity**. In: *Proceedings of the 20th International Symposium on Graph Drawing (GD'12)*. Ed. by Walter Didimo and Maurizio Patrignani. Vol. 7704. Lecture Notes in Computer Science. Springer, 2012, 90–101. DOI: 10.1007/978-3-642-36763-2_9 (see pages 170, 173–175).

[CMS07] Markus Chimani, Petra Mutzel, and Jens M. Schmidt. **Efficient Extraction of Multiple Kuratowski Subdivisions**. In: *Proceedings of the 15th International Symposium on Graph Drawing (GD'07)*. Springer Berlin Heidelberg, 2007, 159–170. DOI: 10.1007/978-3-540-77537-9_17 (see page 166).

[Cor+08] Pier Francesco Cortese, Giuseppe Di Battista, Fabrizio Frati, Maurizio Patrignani, and Maurizio Pizzonia. **C-Planarity of C-Connected Clustered Graphs**. *Journal of Graph Algorithms and Applications* 12:2 (2008), 225–262. DOI: 10.7155/jgaa.00165 (see pages 21, 60, 174, 195).

[Cor+22] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. **Introduction to Algorithms**. Fourth Edition. MIT Press, 2022, 1332. ISBN: 9780262046305 (see page 9).

[CP18] Pier Francesco Cortese and Maurizio Patrignani. **Clustered Planarity = Flat Clustered Planarity**. In: *Proceedings of the 26th International Symposium on Graph Drawing (GD'18)*. Ed. by Therese C. Biedl and Andreas Kerren. Vol. 11282. LNCS. Springer, 2018, 23–38. DOI: 10.1007/978-3-030-04414-5_2 (see page 60).

[DaL+18] Giordano Da Lozzo, Giuseppe Di Battista, Fabrizio Frati, and Maurizio Patrignani. **Computing NodeTrix Representations of Clustered Graphs**. *Journal of Graph Algorithms and Applications* 22:2 (2018), 139–176. DOI: 10.7155/jgaa.00461 (see page 92).

[DaL15] Giordano Da Lozzo. **Planar Graphs with Vertices in Prescribed Regions:models, algorithms, and complexity**. PhD thesis. Roma Tre University, 2015. URL: http://www.dia.uniroma3.it/~dalozzo/files/phd-thesis-giordano-dalozzo.pdf (see pages 1, 2, 17–19, 87, 170).

[Die17] Reinhard Diestel. **Graph Theory**. 5th edition. Graduate Texts in Mathematics. Springer, 2017. ISBN: 3662536218. DOI: 10.1007/978-3-662-53622-3 (see pages 9, 63).

[DLR90] Giuseppe Di Battista, Wei-Ping Liu, and Ivan Rival. **Bipartite Graphs, Upward Drawings, and Planarity**. *Information Processing Letters* 36:6 (1990), 317–322. DOI: 10.1016/0020-0190(90)90045-Y (see page 63).

[DT89] G. Di Battista and R. Tamassia. **Incremental planarity testing**. In: *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS'89)*. IEEE, 1989, 436–441. DOI: 10.1109/sfcs.1989.63515 (see page 103).

[DT90] Giuseppe Di Battista and Roberto Tamassia. **On-line graph algorithms with SPQR-trees**. In: *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*. Springer, 1990, 598–611. DOI: 10.1007/bfb0032061 (see page 103).

[DT96a] Giuseppe Di Battista and Roberto Tamassia. **On-line maintenance of triconnected components with SPQR-trees**. *Algorithmica* 15:4 (1996), 302–318. DOI: 10.1007/bf01961541 (see pages 12, 103, 110).

[DT96b] Giuseppe Di Battista and Roberto Tamassia. **On-Line Planarity Testing**. *SIAM Journal on Computing* 25:5 (Oct. 1996), 956–997. DOI: 10.1137/s0097539794280736 (see pages 33, 103, 110, 120).

[EFK09] Alejandro Estrella-Balderrama, J. Joseph Fowler, and Stephen G. Kobourov. **Graph Simultaneous Embedding Tool, GraphSET**. In: *Proceedings of the 16th International Symposium on Graph Drawing (GD'08)*. Ed. by Ioannis G. Tollis and Maurizio Patrignani. Vol. 5417. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, 169–180. DOI: 10.1007/978-3-642-00219-9_17 (see page 154).

[EFK10] Alejandro Estrella-Balderrama, J. Joseph Fowler, and Stephen G. Kobourov. **GraphSET, a tool for simultaneous graph drawing**. *Software: Practice and Experience* 40:10 (2010), 849–863. DOI: 10.1002/spe.958 (see page 171).

[Elf+01] Matthias Elf, Carsten Gutwenger, Michael Jünger, and Giovanni Rinaldi. **Branch-and-Cut Algorithms for Combinatorial Optimization and Their Implementation in ABACUS**. In: *Computational Combinatorial Optimization: Optimal or Provably Near-Optimal Solutions*. Ed. by Michael Jünger and Denis Naddef. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, 157–222. ISBN: 978-3-540-45586-8. DOI: 10.1007/3-540-45586-8_5 (see page 173).

[Epp+96] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. **Separator Based Sparsification**. *Journal of Computer and System Sciences* 52:1 (1996), 3–27. DOI: 10.1006/jcss.1996.0002 (see pages 103, 104).

[Epp+98] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. **Separator-Based Sparsification II: Edge and Vertex Connectivity**. *SIAM Journal on Computing* 28:1 (1998), 341–381. DOI: 10.1137/S0097539794269072 (see pages 59, 78).

[ET76] Shimon Even and Robert Endre Tarjan. **Computing an st-numbering**. *Theoretical Computer Science* 2:3 (Sept. 1976), 339–344. DOI: 10.1016/0304-3975(76)90086-4 (see pages 11, 23, 31).

[FB04] Michael Forster and Christian Bachmaier. **Clustered Level Planarity**. In: *Proceedings of the 30th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'04)*. Ed. by Peter van Emde Boas, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller. Vol. 2932. Lecture Notes in Computer Science. Springer, 2004, 218–228. DOI: 10.1007/978-3-540-24618-3_18 (see page 87).

[FCE95] Qing-Wen Feng, Robert F. Cohen, and Peter Eades. **Planarity for Clustered Graphs**. In: *Proceedings of the 3rd Annual European Symposium on Algorithms (ESA'95)*. Ed. by Paul G. Spirakis. Vol. 979. LNCS. Springer, 1995, 213–226. DOI: 10.1007/3-540-60313-1_145 (see pages 2, 21, 60).

[Fed+17] Marcus Fedarko, Jay Ghurye, Todd Treagen, and Mihai Pop. **MetagenomeScope: Web-Based Hierarchical Visualization of Metagenome Assembly Graphs**. In: *Proceedings of the 25th International Symposium on Graph Drawing (GD'17)*. Ed. by Fabrizio Frati and Kwan-Liu Ma. (Poster). Springer, 2017, 630–632. DOI: 10.1007/978-3-319-73915-1 (see page 103).

[Feu23] Tim-Florian Feulner. **Deriving Embeddings and Triconnectivity from the Haeupler-Tarjan Planarity Test**. BA thesis. University of Passau, 2023 (see pages 163, 164).

[FMR06] Hubert De Fraysseix, Patrice Ossona De Mendez, and Pierre Rosenstiehl. **Trémaux trees and planarity**. *International Journal of Foundations of Computer Science* 17:05 (2006), 1017–1029. DOI: 10.1142/S0129054106004248 (see pages 2, 166).

[FOO05] D. Franken, J. Ochs, and K. Ochs. **Generation of wave digital structures for networks containing multiport elements**. *IEEE Transactions on Circuits and Systems I: Regular Papers* 52:3 (2005), 586–596. DOI: 10.1109/tcsi.2004.843056 (see page 103).

[Fre22] Valentin Frey. **Planarity Testing and Embedding based on the Haeupler-Tarjan Algorithm: Implemention and Experiments**. BA thesis. University of Passau, 2022 (see pages 163, 164).

[FT22] Radoslav Fulek and Csaba D. Tóth. **Atomic Embeddability, Clustered Planarity, and Thickenability**. *Journal of the ACM* 69:2 (2022), 13:1–13:34. DOI: 10.1145/3502264. arXiv: 1907.13086v1 [cs.CG] (see pages 2, 6, 17, 21, 57–60, 69, 73, 79, 101, 102, 105).

[Ful+12] Radoslav Fulek, Michael J. Pelsmajer, Marcus Schaefer, and Daniel Štefankovič. **Hanani–Tutte, Monotone Drawings, and Level-Planarity**. In: *Thirty Essays on Geometric Graph Theory*. Springer New York, 2012, 263–287. DOI: 10.1007/978-1-4614-0110-0_14 (see page 171).

[Ful+15] Radoslav Fulek, Jan Kynčl, Igor Malinović, and Dömötör Pálvölgyi. **Clustered Planarity Testing Revisited**. *The Electronic Journal of Combinatorics* 22:4 (2015). DOI: 10.37236/5002 (see pages 21, 60, 174).

[Gas+06] Elisabeth Gassner, Michael Jünger, Merijam Percan, Marcus Schaefer, and Michael Schulz. **Simultaneous Graph Embeddings with Fixed Edges**. In: *Graph-Theoretic Concepts in Computer Science*. Springer Berlin Heidelberg, 2006, 325–335. DOI: 10.1007/11917496_29 (see page 21).

[GKM08] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. **Planarity Testing and Optimal Edge Insertion with Embedding Constraints**. *Journal of Graph Algorithms and Applications* 12:1 (2008), 73–95. DOI: 10.7155/jgaa.00160 (see pages 3, 20, 55, 59, 61, 62, 90).

[GM00] Carsten Gutwenger and Petra Mutzel. **A Linear Time Implementation of SPQR-Trees**. In: *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*. Ed. by Joe Marks. Vol. 1984. LNCS. Springer, 2000, 77–90. DOI: 10.1007/3-540-44541-2_8 (see pages 12, 119).

[GMM06] Gregory A. Grothaus, Adeel Mufti, and T. M. Murali. **Automatic layout and visualization of biclusters**. *Algorithms for Molecular Biology* 1:1 (2006), 15. DOI: 10.1186/1748-7188-1-15 (see page 153).

[GMS14] Carsten Gutwenger, Petra Mutzel, and Marcus Schaefer. **Practical Experience with Hanani-Tutte for Testing c-Planarity**. In: *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments (ALENEX'14)*. Ed. by Catherine C. McGeoch and Ulrich Meyer. Society for Industrial and Applied Mathematics, 2014, 86–97. DOI: 10.1137/1.9781611973198.9 (see pages 170, 173–175, 178).

[GT88] Andrew V. Goldberg and Robert Endre Tarjan. **A new approach to the maximum-flow problem**. *Journal of the ACM* 35:4 (1988), 921–940. DOI: 10.1145/48014.61051 (see pages 169, 197).

[Gut+02] Carsten Gutwenger, Michael Jünger, Sebastian Leipert, Petra Mutzel, Merijam Percan, and René Weiskircher. **Advances in C-Planarity Testing of Clustered Graphs**. In: *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*. Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, 220–235. DOI: 10.1007/3-540-36151-0_21 (see pages 21, 60).

[Gut10] Carsten Gutwenger. **Application of SPQR-trees in the planarization approach for drawing graphs**. PhD thesis. Technische Universität Dortmund, 2010. URL: https://eldorado.tu-dortmund.de/bitstream/2003/27430/1/diss_gutwenger.pdf (see page 103).

[Had75] F. Hadlock. **Finding a Maximum Cut of a Planar Graph in Polynomial Time**. *SIAM Journal on Computing* 4:3 (1975), 221–225. DOI: 10.1137/0204019 (see page 1).

[Har04] Jon Harris. **JGraphEd – A Java Graph Editor and Graph Drawing Framework**. Tech. rep. Carleton University, School of Computer Science, Comp 5901 Directed Studies, 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.188.5066&rank=1 (see page 154).

[HFM07] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin. **NodeTrix: a Hybrid Visualization of Social Networks**. *IEEE Transactions on Visualization and Computer Graphics* 13:6 (2007), 1302–1309. DOI: 10.1109/tvcg.2007.70582 (see page 92).

[HH07] Martin Harrigan and Patrick Healy. **Practical Level Planarity Testing and Layout with Embedding Constraints**. In: *Proceedings of the 15th International Symposium on Graph Drawing (GD'07)*. Ed. by Seok-Hee Hong, Takao Nishizeki, and Wu Quan. Vol. 4875. Lecture Notes in Computer Science. Springer, 2007, 62–68. DOI: 10.1007/978-3-540-77537-9_9 (see page 171).

[HM03] Wen-Lian Hsu and Ross M. McConnell. **PC trees and circular-ones arrangements**. *Theoretical Computer Science* 296:1 (2003), 99–116. DOI: 10.1016/S0304-3975(02)00435-8 (see pages 8, 23, 26, 27, 31, 51, 52, 132–135, 167).

[HM04] Wen-Lian Hsu and Ross M. McConnell. **PQ Trees, PC Trees, and Planar Graphs**. In: *Handbook of Data Structures and Applications*. Ed. by Dinesh P. Mehta and Sartaj Sahni. Chapman and Hall/CRC, 2004. Chap. 32. DOI: 10.1201/9781420035179.ch32. URL: https://www.cs.colostate.edu/~rmm/pc2.pdf (see pages 23, 26, 27, 31, 51, 52, 132–135, 137, 139–141, 146, 153, 167).

[HN09] Seok-Hee Hong and Hiroshi Nagamochi. **Two-page book embedding and clustered graph planarity**. Tech. rep. TR[2009-004]. Dept. of Applied Mathematics and Physics, University of Kyoto, 2009. URL: https://citeseerx.ist.psu.edu/doc/10.1.1.361.1233 (see pages 21, 170, 195).

[HN18] Seok-Hee Hong and Hiroshi Nagamochi. **Simpler algorithms for testing two-page book embedding of partitioned graphs**. *Theoretical Computer Science* 725 (2018), 79–98. DOI: 10.1016/j.tcs.2015.12.039 (see page 21).

[HR20a] Jacob Holm and Eva Rotenberg. **Fully-dynamic Planarity Testing in Polylogarithmic Time**. In: *Proceedings of the 52nd Annual ACM Symposium on Theory of Computing (STOC'20)*. Ed. by Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy. Vol. abs/1911.03449. ACM, 2020, 167–180. DOI: 10.1145/3357713.3384249 (see pages 79, 103, 127).

[HR20b] Jacob Holm and Eva Rotenberg. **Worst-Case Polylog Incremental SPQR-trees: Embeddings, Planarity, and Triconnectivity**. In: *Proceedings of the 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'20)*. Society for Industrial and Applied Mathematics, 2020, 2378–2397. DOI: 10.1137/1.9781611975994.146 (see pages 79, 103, 127).

[Hsu01] Wen-Lian Hsu. **PC-Trees vs. PQ-Trees**. In: *Proceedings of the 7th Annual International Conference on Computing and Combinatorics (COCOON'01)*. Ed. by Jie Wang. Vol. 2108. Lecture Notes in Computer Science. Springer, 2001, 207–217. DOI: 10.1007/3-540-44679-6_23 (see page 10).

[Hsu03] Wen-Lian Hsu. **An Efficient Implementation of the PC-Tree Algorithm of Shih & Hsu's Planarity Test**. Tech. rep. Institute of Information Science, Academia Sinica, 2003. URL: http://iasl.iis.sinica.edu.tw/webpdf/paper-2003-PLANAR_implementation.pdf (see page 153).

[HT08] Bernhard Haeupler and Robert Endre Tarjan. **Planarity Algorithms via PQ-Trees (Extended Abstract)**. *Electronic Notes in Discrete Mathematics* 31 (2008), 143–149. DOI: 10.1016/j.endm.2008.06.029 (see pages 5, 23, 29, 31, 34, 47, 51, 55, 56, 131, 163, 168, 195).

[HT73a] John Edward Hopcroft and Robert Endre Tarjan. **Algorithm 447: efficient algorithms for graph manipulation**. *Communications of the ACM* 16:6 (June 1973), 372–378. DOI: 10.1145/362248.362272 (see page 31).

[HT73b] John Edward Hopcroft and Robert Endre Tarjan. **Dividing a Graph into Triconnected Components**. *SIAM Journal on Computing* 2:3 (1973), 135–158. DOI: 10.1137/0202012 (see pages 1, 12, 13, 110, 119).

[JKR13] Vít Jelínek, Jan Kratochvíl, and Ignaz Rutter. **A Kuratowski-type theorem for planarity of partially embedded graphs**. *Computational Geometry* 46:4 (May 2013), 466–492. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2012.07.005 (see page 197).

[JLM98] Michael Jünger, Sebastian Leipert, and Petra Mutzel. **Level Planarity Testing in Linear Time**. In: *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*. Ed. by Sue Whitesides. Vol. 1547. Lecture Notes in Computer Science. Springer, 1998, 224–237. DOI: 10.1007/3-540-37623-2_17 (see pages 2, 19).

[JS09] Michael Jünger and Michael Schulz. **Intersection Graphs in Simultaneous Embedding with Fixed Edges**. *Journal of Graph Algorithms and Applications* 13:2 (2009), 205–218. DOI: 10.7155/jgaa.00184 (see pages 57, 88, 98, 99, 102).

[KKV11] Pavel Klavík, Jan Kratochvíl, and Tomás Vyskocil. **Extending Partial Representations of Interval Graphs**. In: *Proceedings of the 8th Annual Conference on Theory and Applications of Models of Computation (TAMC'11)*. Ed. by Mitsunori Ogihara and Jun Tarui. Vol. 6648. Lecture Notes in Computer Science. Springer, 2011, 276–285. DOI: 10.1007/978-3-642-20877-5_28 (see page 33).

[Kla+12] Pavel Klavík, Jan Kratochvíl, Tomasz Krawczyk, and Bartosz Walczak. **Extending Partial Representations of Function Graphs and Permutation Graphs**. In: *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Ed. by Leah Epstein and Paolo Ferragina. Vol. 7501. Lecture Notes in Computer Science. Springer, 2012, 671–682. DOI: 10.1007/978-3-642-33090-2_58 (see page 33).

[Kla+16] Pavel Klavík, Jan Kratochvíl, Yota Otachi, Toshiki Saitoh, and Tomáš Vyskočil. **Extending Partial Representations of Interval Graphs**. *Algorithmica* 78:3 (2016), 945–967. DOI: 10.1007/s00453-016-0186-z (see page 33).

[Kla+17] Pavel Klavík, Jan Kratochvíl, Yota Otachi, Ignaz Rutter, Toshiki Saitoh, Maria Saumell, and Tomás Vyskocil. **Extending Partial Representations of Proper and Unit Interval Graphs**. *Algorithmica* 77:4 (2017), 1071–1104. DOI: 10.1007/s00453-016-0133-z (see page 33).

[KS18] Pavel Klavik and Maria Saumell. **Minimal Obstructions for Partial Representations of Interval Graphs**. *The Electronic Journal of Combinatorics* 25:4 (Dec. 2018). ISSN: 1077-8926. DOI: 10.37236/5862 (see page 197).

[KW17] Tomasz Krawczyk and Bartosz Walczak. **Extending Partial Representations of Trapezoid Graphs**. In: *Proceedings of the 43rd Workshop on Graph-Theoretic Concepts in Computer Science (WG'17)*. Ed. by Hans L. Bodlaender and Gerhard J. Woeginger. Vol. 10520. Lecture Notes in Computer Science. Springer, 2017, 358–371. DOI: 10.1007/978-3-319-68705-6_27 (see page 33).

[LB62] C. Lekkeikerker and J. Boland. **Representation of a finite graph by a set of intervals on the real line**. *Fundamenta Mathematicae* 51:1 (1962), 45–64. ISSN: 1730-6329. DOI: 10.4064/fm-51-1-45-64 (see page 197).

[LEC67] A. Lempel, S. Even, and I. Cederbaum. **An algorithm for planarity testing of graphs**. *Theory of Graphs* (1967). Ed. by P. Rosenstiehl, 215–232 (see page 23).

[Lei97] Sebastian Leipert. **PQ-Trees, An Implementation as Template Class in C++**. Tech. rep. University of Cologne, 1997. URL: http://e-archive.informatik.uni-koeln.de/id/eprint/259 (see pages 8, 153).

[Lei98] Sebastian Leipert. **Level planarity testing and embedding in linear time**. PhD thesis. Universität zu Köln, 1998 (see page 171).

[Len89] Thomas Lengauer. **Hierarchical Planarity Testing Algorithms**. *Journal of the ACM* 36:3 (1989), 474–509. DOI: 10.1145/65950.65952 (see pages 2, 20, 21, 60).

[LMM18] Anna Lubiw, Tillmann Miltzow, and Debajyoti Mondal. **The Complexity of Drawing a Graph in a Polygonal Region**. In: *Proceedings of the 26th International Symposium on Graph Drawing (GD'18)*. Ed. by Therese Biedl and Andreas Kerren. Vol. 11282. Lecture Notes in Computer Science. Springer, 2018, 387–401. DOI: 10.1007/978-3-030-04414-5_28 (see page 33).

[LRT21] Giuseppe Liotta, Ignaz Rutter, and Alessandra Tappini. **Simultaneous FPQ-ordering and hybrid planarity testing**. *Theoretical Computer Science* (2021). DOI: 10.1016/j.tcs.2021.05.012 (see pages 61, 92, 94, 95).

[Mac37] Saunders Mac Lane. **A structural characterization of planar combinatorial graphs**. *Duke Mathematical Journal* 3:3 (1937), 460–472. DOI: 10.1215/S0012-7094-37-00336-3 (see page 110).

[MS12] A. von Manteuffel and C. Studerus. **Reduze 2 - Distributed Feynman Integral Reduction**. 2012. arXiv: 1201.4330 (see page 103).

[Mut03] Petra Mutzel. **The SPQR-Tree Data Structure in Graph Drawing**. In: *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*. Ed. by Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger. Vol. 2719. LNCS. Springer, 2003, 34–46. DOI: 10.1007/3-540-45061-0_4 (see page 103).

[Neu68] L. Neuwirth. **An Algorithm for the Construction of 3-Manifolds from 2-Complexes**. *Mathematical Proceedings of the Cambridge Philosophical Society* 64:3 (1968), 603–614. DOI: 10.1017/S0305004100043279 (see page 60).

[Opa79] J. Opatrny. **Total Ordering Problem**. *SIAM Journal on Computing* 8:1 (Feb. 1979), 111–114. DOI: 10.1137/0208008 (see pages 98, 99).

[PAC02] Helen C. Purchase, Jo-Anne Allder, and David Carrington. **Graph Layout Aesthetics in UML Diagrams: User Preferences**. *Journal of Graph Algorithms and Applications* 6:3 (2002), 255–279. DOI: 10.7155/jgaa.00054 (see page 1).

[Pat06] Maurizio Patrignani. **On Extending a Partial Straight-line Drawing**. *International Journal of Foundations of Computer Science* 17:5 (2006), 1061–1070. DOI: 10.1142/S0129054106004261 (see pages 2, 33).

[Pat13] Maurizio Patrignani. **Planarity Testing and Embedding**. In: *Handbook of Graph Drawing and Visualization*. Ed. by Roberto Tamassia. Chapman and Hall/CRC, 2013. Chap. 1, 1–42. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/planarity.pdf (see pages 1, 2, 9, 23, 33, 173).

[Pfr20] Matthias Pfretzschner. **A Linear-Time Implementation of PC-trees**. BA thesis. University of Passau, 2020. URL: https://www.fim.uni-passau.de/fileadmin/dokumente/fakultaeten/fim/lehrstuhl/rutter/abschlussarbeiten/ba-pfretzschner.pdf (see pages 27, 69).

[Pou92] J. A. La Poutré. **Maintenance of triconnected components of graphs**. In: *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92)*. Springer, 1992, 354–365. DOI: 10.1007/3-540-55719-9_87 (see page 103).

[Pou94] Johannes A. La Poutré. **Alpha-algorithms for incremental planarity testing (preliminary version)**. In: *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC'94)*. ACM Press, 1994. DOI: 10.1145/195058.195439 (see page 103).

[Rad20] Marcel Radermacher. **Geometric Graph Drawing Algorithms - Theory, Engineering and Experiments**. PhD thesis. Karlsruher Institut für Technologie (KIT), 2020. 227 pp. DOI: 10.5445/IR/1000117664 (see page 188).

[Ran+01] Bert Randerath, Ewald Speckenmeyer, Endre Boros, Peter L. Hammer, Alexander Kogan, Kazuhisa Makino, Bruno Simeone, and Ondrej Cepek. **A Satisfiability Formulation of Problems on Level Graphs**. *Electronic Notes in Discrete Mathematics* 9 (2001), 269–277. DOI: 10.1016/S1571-0653(04)00327-0 (see page 171).

[Rut20] Ignaz Rutter. **Simultaneous Embedding**. In: *Beyond Planar Graphs*. Ed. by Seok-Hee Hong and Takeshi Tokuyama. Springer Singapore, 2020. Chap. 13, 237–265. DOI: 10.1007/978-981-15-6533-5_13 (see pages 2, 21, 98).

[Sch13] Marcus Schaefer. **Toward a Theory of Planarity: Hanani-Tutte and Planarity Variants**. *Journal of Graph Algorithms and Applications* 17:4 (2013), 367–440. DOI: 10.7155/jgaa.00298 (see pages 1–3, 17–21, 55, 81, 85, 170).

[SH99] Wei-Kuan Shih and Wen-Lian Hsu. **A new planarity test**. *Theoretical Computer Science* 223:1-2 (1999), 179–191. DOI: 10.1016/s0304-3975(98)00120-0 (see page 131).

[TL84] Robert E. Tarjan and Jan van Leeuwen. **Worst-case Analysis of Set Union Algorithms**. *Journal of the ACM* 31:2 (Mar. 1984), 245–281. ISSN: 1557-735X. DOI: 10.1145/62.2160 (see pages 121, 135, 167).

[VVK09] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. **The refined process structure tree**. *Data and Knowledge Engineering* 68:9 (2009), 793–818. DOI: 10.1016/j.datak.2009.02.015 (see page 103).

[War+02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. **Cognitive Measurements of Graph Aesthetics**. *Information Visualization* 1:2 (June 2002), 103–110. DOI: 10.1057/palgrave.ivs.9500013 (see page 1).

[Wei02] Rene Weiskircher. **New applications of SPQR-trees in graph drawing**. en. PhD thesis. Universität des Saarlandes, 2002. DOI: 10.22028/D291-25752 (see page 103).

[Wes92] Jeffery Westbrook. **Fast incremental planarity testing**. In: *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92)*. Springer, 1992, 342–353. DOI: 10.1007/3-540-55719-9_86 (see page 103).

[Zan12] João Paulo Pereira Zanetti. **Complexidade de construção de árvores PQR**. MA thesis. Universidade Estadual de Campinas, Instituto de Computação, 2012. URL: http://bdtd.ibict.br/vufind/Record/CAMP_0b551865d78ef032289f17f95e3ccee7 (see page 154).

[Zha+13] Ye Zhang, Wai-Shing Luk, Hai Zhou, Changhao Yan, and Xuan Zeng. **Layout decomposition with pairwise coloring for multiple patterning lithography**. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*. Ed. by Jörg Henkel. IEEE, 2013, 170–177. DOI: 10.1109/ICCAD.2013.6691115 (see page 103).

List of Publications

Articles in Refereed Journals

[1] **Synchronized Planarity with Applications to Constrained Planarity Problems**. *ACM Transactions on Algorithms* (2023). DOI: 10.1145/3607474 (see page 57). Joint work with Thomas Bläsius and Ignaz Rutter.

[2] **Experimental Comparison of PC-Trees and PQ-Trees**. *Journal of Experimental Algorithmics* (2023). DOI: 10.1145/3611653 (see page 131). Joint work with Matthias Pfretzschner and Ignaz Rutter.

Articles in Refereed Conference Proceedings

[3] **Synchronized Planarity with Applications to Constrained Planarity Problems**. In: *Proceedings of the 29th Annual European Symposium on Algorithms (ESA'21)*. Ed. by Petra Mutzel, Rasmus Pagh, and Grzegorz Herman. Vol. 204. Leibniz International Proceedings in Informatics. Schloss Dagstuhl – Leibniz Center for Informatics, 2021, 19:1–19:14. DOI: 10.4230/LIPIcs.ESA.2021.19 (see page 57). Joint work with Thomas Bläsius and Ignaz Rutter.

[4] **Experimental Comparison of PC-Trees and PQ-Trees**. In: *Proceedings of the 29th Annual European Symposium on Algorithms (ESA'21)*. Ed. by Petra Mutzel, Rasmus Pagh, and Grzegorz Herman. Vol. 204. Leibniz International Proceedings in Informatics. Schloss Dagstuhl – Leibniz Center for Informatics, 2021, 43:1–43:13. DOI: 10.4230/LIPIcs.ESA.2021.43 (see page 131). Joint work with Matthias Pfretzschner and Ignaz Rutter.

[5] **Parameterized Complexity of Simultaneous Planarity**. In: *Proceedings of the 31st International Symposium on Graph Drawing (GD'23)*. Ed. by Michael A. Bekos and Markus Chimani. Vol. 14466. Lecture Notes in Computer Science. Springer-Verlag, 2023, 82–96. DOI: 10.1007/978-3-031-49275-4_6 (see page 99). Joint work with Matthias Pfretzschner and Ignaz Rutter.

[6] **Maintaining Triconnected Components Under Node Expansion**. In: *Proceedings of the 13th Conference on Algorithms and Complexity (CIAC'23)*. Ed. by Marios Mavronicolas. Vol. 13898. Lecture Notes in Computer Science. Springer-Verlag, 2023, 202–216. DOI: 10.1007/978-3-031-30448-4_15 (see page 103). Joint work with Ignaz Rutter.

[7] **Constrained Planarity in Practice – Engineering the Synchronized Planarity Algorithm**. In: *Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALENEX'24)*. Ed. by Rezaul Chowdhury and Solon P. Pissis. 2024, 1–14. DOI: 10.1137/1.9781611977929.1 (see page 169). Joint work with Ignaz Rutter.

Further Articles

[8] **A Simple Partially Embedded Planarity Test Based on Vertex-Addition**. In: *currently under review*. 2023 (see pages 23, 33). Joint work with Ignaz Rutter and Sandhya T. P..

[9] **Maintaining Triconnected Components Under Node Expansion**. In: *Proceedings of the 39th European Workshop on Computational Geometry (EuroCG'23)*. 2023. URL: https://dccg.upc.edu/eurocg23/wp-content/uploads/2023/05/Booklet_EuroCG2023.pdf#page=159 (see page 103). Joint work with Ignaz Rutter.

Poster

[10] **odgf-python – A Python Interface for the Open Graph Drawing Framework**. In: *Proceedings of the 31st International Symposium on Graph Drawing (GD'23)*. to appear. 2023. Joint work with Andreas Strobl.