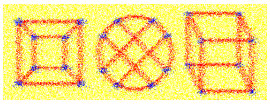

Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 29, no. 1, pp. 91–123 (2025)
DOI: 10.7155/jgaa.v29i1.2923

Constrained Planarity in Practice Engineering the Synchronized Planarity Algorithm

Simon D. Fink[1] *Ignaz Rutter*[2]

[1]Technische Universität Wien, Algorithms and Complexity Group, Austria
[2]University of Passau, Faculty of Computer Science and Mathematics, Germany

Submitted: May 2024 Accepted: May 2025 Published: May 2025

Article type: Regular Communicated by: Markus Chimani

Abstract. In the constrained planarity setting, we ask whether a graph admits a planar drawing that additionally satisfies a given set of constraints. These constraints are often derived from very natural problems; prominent examples are Level Planarity, where vertices have to lie on given horizontal lines indicating a hierarchy, and Clustered Planarity, where we in addition to the graph itself draw the boundaries of clusters which recursively group the vertices in a crossing-free manner. Despite receiving significant amount of attention and substantial theoretical progress on these problems, only very few of the found solutions have been put into practice and evaluated experimentally.

In this paper, we describe our implementation of the recent quadratic-time algorithm by Bläsius et al. [8] for solving the problem Synchronized Planarity, which can be seen as a common generalization of several constrained planarity problems, including the aforementioned ones. 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.

Funded by the Deutsche Forschungsgemeinschaft (German Research Foundation, DFG) under grant RU-1903/3-1 and by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT22029].

A preliminary version of this paper has appeared as S. D. Fink, I. Rutter, *Constrained Planarity in Practice – Engineering the Synchronized Planarity Algorithm* in Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX 2024), pages 1–14, 2024, doi:10.1137/1.9781611977929.1.

Our source code is available at github.com/N-Coder/syncplan.

E-mail addresses: sfink@ac.tuwien.ac.at (Simon D. Fink) rutter@fim.uni-passau.de (Ignaz Rutter)

1 Introduction

In many practical graph drawing applications we not only seek any 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 changes to a graph are shown in steps while keeping a static part fixed. In practice, clustered drawings are for example 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. As crossings negatively affect the readability of drawings [46, 51], we preferably seek planar, i.e. crossing-free, drawings. The combination of these concepts leads to the field of constrained planarity problems, where we ask whether a graph admits a planar drawing that satisfies a given set of constraints. This includes the problems Level Planarity [42, 16], Clustered Planarity [11, 44, 27], and Simultaneous Embedding with Fixed Edges (SEFE) [15, 9, 49], which respectively model the aforementioned applications; see Figure 1. Formally, these problems are defined as follows.

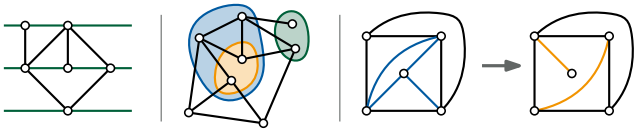


Figure 1: Examples of constrained planarity problems: Level Planarity (a), Clustered Planarity (b), SEFE (c).

Problem Level Planarity

Given graph G, leveling function $\ell : V(G) \to \mathbb{N}$
Question Is there a planar drawing where each vertex $v \in V(G)$ has y-coordinate $\ell(v)$ and all edges are drawn y-monotone?

Problem Clustered Planarity

Given graph G, rooted cluster tree T, cluster assignment function $\gamma : V(G) \to V(T)$
Question Is there a planar drawing where, for each cluster $c \in V(T)$, we can add a simple closed region that
1. encloses exactly the vertices mapped to c or one of its descendants in T, and
2. has a border that crosses each edge that connects a vertex within its interior to a vertex on its outside exactly once, but no other edge or cluster region border?

Problem SEFE

Given graphs $G^{①}, G^{②}$ with a shared graph $G = G^{①} \cap G^{②}$
Question Are there planar drawings of $G^{①}$ and $G^{②}$ that induce the same drawing of their shared part G?

In the last years, constrained planarity problems, which include the ones above, 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 [50] and [22] for an overview. 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 [45], which have also been thoroughly assessed in terms of their practical running time [31, 14].

The recently introduced problem Synchronized Planarity [8] 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. Akin to the Goldberg and Tarjan push-relabel algorithm [36], it uses few and simple operations that can be applied in arbitrary order. Through reductions from many other problems (see Figure 3 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 paper, 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 previous theoretical running time analysis by Bläsius et al. [8] 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 3 provides more background on constrained planarity and Synchronized Planarity in particular, as well as giving an overview of previous practical approaches to constrained planarity. In Section 4 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 even on large instances in Section 5. We analyze the effects of our engineering in greater detail in Section 6.

2 Preliminaries

We rely on some well-known concepts from the fields of graph drawing and planar graphs. We only briefly define the most important terms here and refer to the theoretical description of the implemented algorithm [8] for more comprehensive definitions. A more gentle introduction to the concepts can also be found in Chapter 1 of the Handbook of Graph Drawing and Visualization [45].

We consider two planar (i.e., crossing-free) drawings equivalent if they define the same *rotation system*, which specifies for each vertex its *rotation*, i.e., the cyclic order of the edges around the vertex. An *embedding* is an equivalence class of planar drawings induced by this relation. An *embedding tree* [8] is a PQ-tree [13] that describes all possible rotations of a vertex in a planar graph; see Figure 2d. Its leaves correspond to the incident edges, while its inner nodes are either Q-nodes, which dictate a fixed ordering of their incident subtrees that can only be reversed, or are P-nodes, which allow arbitrary permutation.

A BC-tree describes the decomposition of a connected graph into its *biconnected* components, which cannot be disconnected by the removal of a so-called *cut-vertex*. Each node of a BC-tree represents either a cut-vertex or a maximal biconnected *block*. We refer to a vertex that is not a cut-vertex as *block-vertex*. An SPQR-tree [24] describes the decomposition of a biconnected

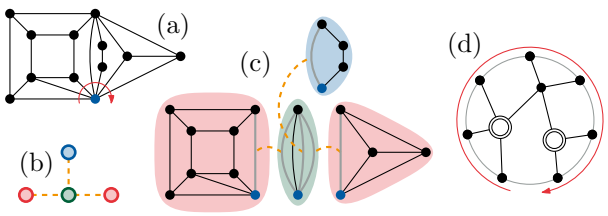


Figure 2: A planar graph (a), its SPQR-tree (b) and the corresponding skeletons (c). Rigids are highlighted in red, parallels in green, and series in blue. The embedding tree of the vertex marked in blue (d). Small black disks are P-nodes, larger white disks are Q-nodes.

graph into its *triconnected* components, which cannot be disconnected by the removal of a so-called *split-pair* of two vertices. Each inner node represents a *skeleton*, which is either a triconnected minor whose planar embedding can only be mirrored (referred to as *rigid* skeleton), a split-pair of two *pole* vertices connected by multiple subgraphs that can be permuted arbitrarily (called *parallel* skeleton), or a cycle formed by split-pairs separating a cyclic sequence of subgraphs (called *series* skeleton); see Figure 2c.

See also [30] for more details on embedding trees and SPQR-trees and their usage in the context of Synchronized Planarity. All three kinds of trees can be computed in time linear in the size of the given graph [12, 45, 38].

3 Related Work

In this section, we first give an overview over the historical development and conceptual relationships of the main constrained planarity variants we consider in this paper. In addition to explaining the theoretical framework of these relationships, Section 3.1 gives more background on the problems SEFE and Clustered Planarity. Section 3.2 is dedicated to the Synchronized Planarity problem and explains its quadratic solution by Bläsius et al. [8]. Finally, Section 3.3 gives an overview over previous practical approaches to solving constrained planarity problems.

3.1 Constrained Planarity Problems

Schaefer [50, Figure 2] introduced a hierarchy on the various variants of constrained planarity that have been studied in the past. Figure 3 shows a subset of this hierarchy, incorporating updates up to 2015 by Da Lozzo [22, Figure 0.1]. Arrows indicate that the target problem either generalizes the source problem or solves it via a reduction. In the version of Da Lozzo, the problems Strip, Clustered and Synchronized Planarity as well as (Connected) SEFE 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 [34]. The only problem from this hierarchy that remains with an unknown complexity is SEFE. In this section, we want to give a short summary of the history of Clustered Planarity and SEFE, which we see central to the field of constrained planarity and which also serve as a motivation for Synchronized Planarity. Afterwards, we will give a short summary of the algorithm we implement for solving the latter problem. We point the interested reader to the original description [8] for full details.

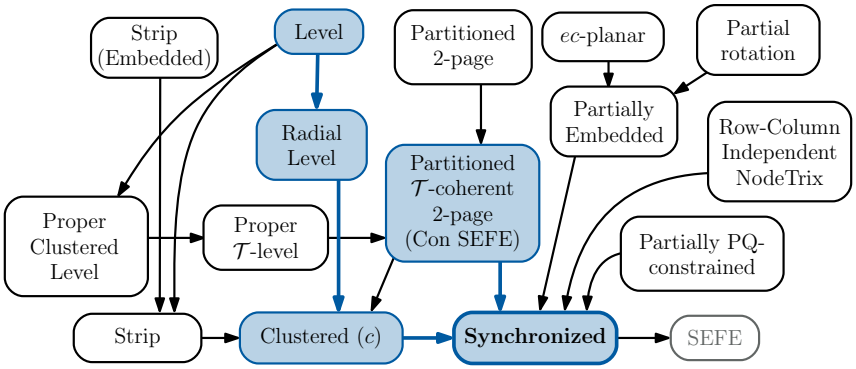


Figure 3: Constrained planarity variants related to Synchronized Planarity, updated selection from [22]. Problems and reductions marked in blue are used for generating test instances.

Recall that in SEFE, we are given two graphs that share some common part and we want to embed both graphs individually such that their common parts are embedded the same way [15, 9, 49]. More general SEFE variants are often NP-complete, e.g., the case with three given graphs [35], even if all share the same common part [3, 50]. 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 [6, 10, 5]. The case where the shared graph is connected, which is called Connected SEFE, was shown to be equivalent to the so-called Partitioned $\mathcal{T}$-coherent 2-page Book Embedding problem [6] and to be reducable to Clustered Planarity [1], all of which were recently shown to be efficiently solvable [34]. In contrast to these results, the complexity of the general SEFE problem with two graphs sharing an arbitrary common graph is still unknown.

Recall that in Clustered Planarity, the embedding has to respect a laminar family of clusters, that is every vertex is assigned to some (hierarchically nested) cluster and an edge may only cross a the border of a cluster's region if it connects a vertex from the inside with one from the outside [11, 44]; see Figure 4a for an example. Lengauer [44] studied and solved this problem as early as 1989 in the setting where the clusters are connected. Feng et al. [27], who coined the term Clustered Planarity, rediscovered this algorithm and asked the general question where disconnected clusters are allowed. This question remained open for 25 years. In that time, polynomial-time algorithms were found for many special-cases [2, 21, 20, 23, 32, 37] before Fulek and Tóth [34] 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. Shortly thereafter, Bläsius et al. [8] gave a solution with running time in $O((n+d)^2)$ that works via a linear-time reduction to Synchronized Planarity.

3.2 Synchronized Planarity

In Synchronized Planarity, we are given a graph together with a set of *pipes*, each of which pairs up two distinct vertices of the graph. Each pipe synchronizes the rotation of its two paired-up vertices (its *endpoints*) in the following sense: We seek a planar embedding of the graph where for each pipe ρ, the rotations of its endpoints u, v line up under the bijection φ_ρ associated with ρ [8]. This ensures that, in any solution, we maintain planarity when we "join" and remove ρ (together with u and v) by identifying the edges incident to u and v according to φ_ρ. See Figure 4c for an example instance of Synchronized Planarity, where joining all pipes again yields the graph of Figure 4a (but in the process looses all clustering information). Formally, this problem is defined as follows.

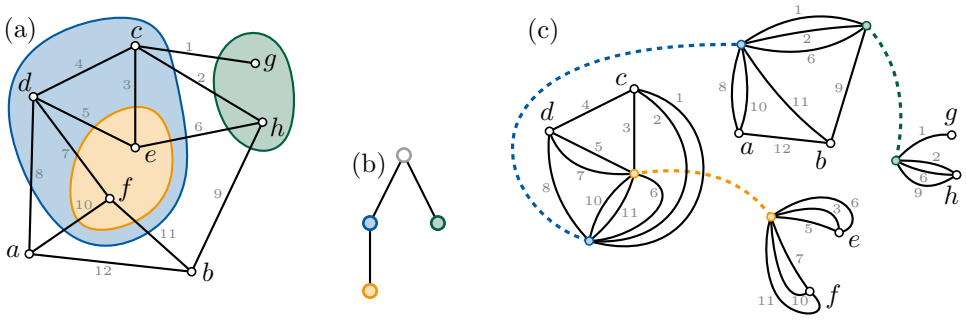


Figure 4: A Clustered Planarity instance (a), its cluster tree (b), and its CD-tree representation (c), which can also be interpreted as an instance of Synchronized Planarity with dashed edges representing pipes whose bijections map edges with the same label to each other. To obtain the component containing e, f in (c) from the clustered graph in (a), the complement of the orange cluster was contracted into the orange vertex. The order of the edges around this orange vertex now corresponds to the (reversed) order of edges leaving the orange cluster. The component of d, c was obtained by separately contracting all vertices in the orange cluster into the orange vertex and all vertices *not* within the blue cluster into the blue vertex. The component of a, b was obtain by separately contracting all child clusters of the root cluster.

Problem Synchronized Planarity[a]

Given graph G and a set $\mathcal{P}$, where each *pipe* $\rho \in \mathcal{P}$ consists of two distinct vertices $v_1, v_2 \in V(G)$ and a bijection φ_ρ between the edges incident to v_1 and those incident to v_2, and each vertex is part of at most one pipe

Question Is there a drawing of G where for each pipe $\rho = (v_1, v_2, \varphi_\rho)$, applying bijection φ_ρ to each element of the cyclic order of edges incident to v_1 yields the cyclic order of edges incident to v_2?

[a]Note that we disregard the originally included Q-vertices here, as they can also be modeled using pipes [8, Section 5].

The motivation for this "synchronization" can best be seen by considering the reduction from Clustered to Synchronized Planarity. At each cluster boundary, we split the graph into two halves: one where we contract the inside of the cluster into a single vertex and one where we contract the outside into a single vertex. In a clustered planar embedding, the order of the edges "leaving" one cluster (i.e. the rotation of its contracted vertex in the one half) needs to match the order in which they "enter" the parent cluster (i.e. the the rotation of the corresponding contracted vertex in the other half). This graph resulting from separately contracting each side of a cluster boundary is called CD-tree [11]; see Figure 4c and [8, Figure 6] for an example. Using this graph, the synchronization of rotations can easily be modeled via Synchronized Planarity by pairing the two contracted vertices corresponding to the same cluster boundary with a pipe. Without this synchronization, one would effectively embed all clusters separately without a guarantee that the separate embeddings agree on the cyclic order in which edges cross cluster boundaries, i.e., that the separate embeddings can be combined to a whole one.

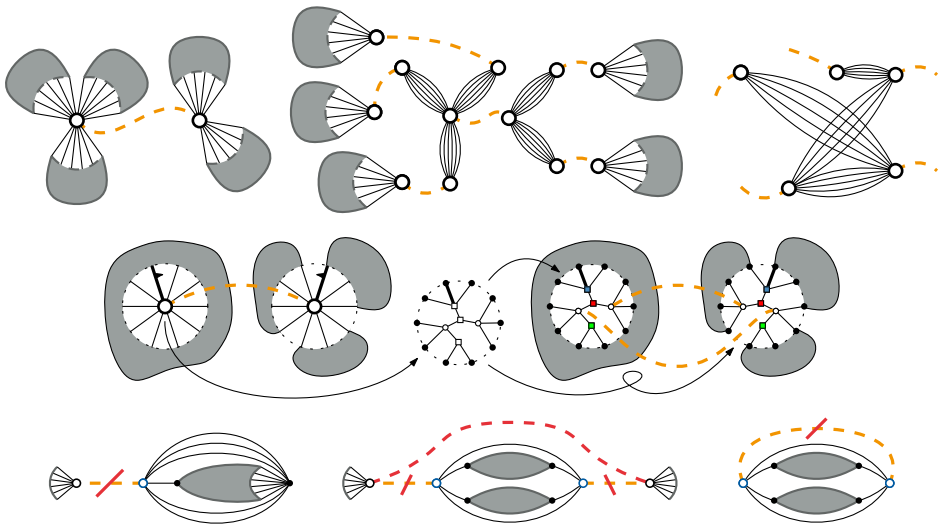


Figure 5: The operations for solving Synchronized Planarity [8], Figure from [30]. Pipes are indicated by orange dashed lines, their endpoints are shown as larger disks. **Top:** Two cut-vertices paired-up by a pipe (left), the result of encapsulating their incident blocks (middle) and the bipartite graph resulting from joining both cut-vertices (right). **Middle:** A block-vertex pipe endpoint (left) that has a non-trivial embedding tree (middle) that is propagated to replace both the vertex and its partner (right). **Bottom,** from left to right: The terminal, transitive, and toroidal case of paired-up vertices with trivial embedding trees (blue) and how their pipes can be removed or replaced (red).

In the quadratic algorithm for solving Synchronized Planarity, a pipe is *feasible* if one of the three following operation can be applied to remove it.

`EncapsulateAndJoin` If both endpoints of the pipe are cut-vertices, they are "encapsulated" by collapsing each incident block to a single vertex to obtain two stars with paired-up centers. Additionally, we split the original components at the two cut-vertices, so that each of their incident blocks is retained as separate component with its own copy of the cut-vertex. These copies are synchronized with the respective vertex incident to the original cut-vertex representing the collapsed block. Now the cut-vertices can be removed by "joining" both stars at their centers, i.e, by identifying their incident edges according to the given bijection; see the top row of Figure 5.

`PropagatePQ` If one endpoint of the pipe is a block-vertex and has an embedding tree that not only consists of a single P-node (i.e., it is *non-trivial*), a copy of this embedding tree is inserted ("propagated") in place of each respective pipe endpoint. The inner nodes of the embedding trees are synchronized by pairing corresponding vertices with a pipe; see the middle row of Figure 5. Note that, as Q-nodes only have a binary embedding decision, they can also easily be synchronized via a 2-SAT formula instead of using pipes.

`SimplifyMatching` In the remaining case, at least one of the endpoints of the pipe is a block-vertex but has a trivial embedding tree. If the vertex (or, more precisely, the parallel skeleton in the SPQR-tree that completely defines its rotation) can respect arbitrary rotations, we can simply remove the pipe. We call this the terminal case. When the other pole of the parallel is

also paired-up and has a trivial embedding tree (the transitive case), we "short-circuit" the pipe across the parallel; see the bottom row of Figure 5. The only exception is if the pipe matches the poles of the same parallel (the toroidal case), where we can again remove the pipe without replacement or directly report a no-instance based on a simple check.

The algorithm then works by simply applying a suitable operation on an *arbitrary* feasible pipe each step. 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 [8, Lemma 3.5]. Especially, this means that maximum-degree pipes are always feasible.

Each of the three operations runs in time linear in the degree of the removed pipe once the embedding trees it depends on have been computed. This is dominated by the time spent on computing the embedding tree, which is linear in the size of the considered biconnected component. Every applied operation removes a pipe, but potentially introduces new pipes of smaller degree. Bläsius et al. [8] show that the progress made by the removal of a pipe always dominates the overhead of the newly-introduced pipes and that the number of operations needed to remove all pipes is limited by the total degree of all paired-up vertices. Furthermore, the resulting instance without pipes can be solved and embedded in linear time. An embedding of the input graph can then be obtained by undoing all changes made to the graph in reverse order while maintaining the found embedding. The algorithm thus runs in the following three simple phases:

1. While pipes are left, choose and remove an arbitrary feasible pipe by applying an operation.
2. Solve and embed the resulting pipe-free (*reduced*) instance.
3. Undo all applied operations while maintaining the embedding.

To gain more control over the order in which pipes are processed, we use a priority queue-based approach for the pipe removal of phase (1). This approach is shown in Algorithm 1 and works along the case distinction showing that there always is a feasible pipe, and thereby that all pipes can be removed by a sequence of operations [8, Lemma 3.5]. We will use different ways of assigning priority to pipes, e.g. by descending degree. Note that in Line 4 and Line 5, we detect that the currently processed pipe is not feasible; see Figure 6. As we cannot process the current pipe in this case, we instead increase the priority of (or directly process) the pipe ρ' that blocks the current pipe ρ from being feasible. Note that, in the former case, we always have $\deg(\rho') > \deg(\rho)$ [8, Lemma 3.5], so this cannot introduce infinite loops. This is also why processing pipes by decreasing degree ensures that the current pipe is always feasible, and thus changing priorities is only needed when using any other processing order.

3.3 Related Practical 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 Clus-

Figure 6: Pipe ρ' blocking the current pipe ρ from being feasible in Line 4 (left) and Line 5 (right) of Algorithm 1.

Algorithm 1: Priority queue-based algorithm for removing all pipes.

```
while !pipes.empty() do
    ρ ← pipes.pop();
    if ρ is cut-cut then
1       EncapsulateAndJoin(ρ); continue;
    T ← embedding tree of block end x of ρ;
    if T non-trivial then
2       PropagatePQ(ρ, T); continue;
    x′ ← other pole of P-node of x;
    if x′ unpaired then
3       SimplifyMatching(ρ); continue ; // terminal case
    ρ′ ← pipe with endpoint x′;
    if x′ is cut-vertex then
4       pipes.push(ρ); increasePriority(ρ′); continue;
    if embedding tree T′ of x′ is non-trivial then
5       pipes.push(ρ); PropagatePQ(ρ′, T′); continue;
    if ρ = ρ′ then
6       SimplifyMatching(ρ); // toroidal case
    else
7       SimplifyMatching(ρ); // transitive case
```

TERED PLANARITY were studied the most, although all implementations predate the first polynomial-time solution and thus either have an exponential worst-case running time or cannot solve all instances. Chimani et al. [17] studied the problem of finding maximal cluster planar subgraphs in practice using an Integer Linear Program (ILP) together with a branch-and-cut algorithm, thereby also obtaining the first ever practical test for c-planarity for general graphs. A later work [19] strengthened the ILP for the special case of testing CLUSTERED PLANARITY, further improving the practical running time. The work by Gutwenger et al. [39] takes a different approach by using a Hanani-Tutte-style formulation of the problem based on the work by Schaefer [50]. 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 [39].

The only other constrained planarity variant for which we could find experimental results is PARTITIONED 2-PAGE BOOK EMBEDDING. Angelini et al. [4] describe an implementation of the SPQR-tree-based linear-time algorithm by Hong and Nagamochi [41], which solves instances with up to 100 000 vertices and two clusters in up to 40 seconds. Unfortunately, their implementation is not publicly available. For (RADIAL) LEVEL PLANARITY, prototypical implementations were described in the dissertations by Leipert [43] and Bachmaier [7], 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 [16]. Simpler algorithms with a super-linear running time have been proposed [40, 48, 33]. For these, we could only find an implementation by Estrella-Balderrama et al. [26] for the quadratic algorithm by

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)
LVL-LRG	1103	$\leq 10^5$ (25 442.0)	0.9–2.0 (1.4)	=1	≤20 000 (1412.5)	≤103 401 (24 779.5)

Table 1: Statistics for our different Synchronized Planarity 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.

Harrigan and Healy [40]. 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 3) 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.

4 Clustered Planarity in Practice

In this section, we shortly describe our C++ implementation of the Synchronized Planarity algorithm by Bläsius et al. [8] and compare its running time and results on instances derived from Clustered Planarity with those of the two existing implementations by Chimani et al. [17, 19] and by Gutwenger et al. [39]. We base our implementation on the graph data structures provided by the OGDF [18] and, as the only other dependency, use the PC-tree implementation by Fink et al. [29] for the embedding trees. The PC-tree is a datastructure that is conceptually equivalent to the PQ-tree we use as embedding tree, but is faster in practice [29].

The algorithm for Synchronized Planarity makes no restriction on how the next feasible pipe should be chosen. For now, we will process the pipes by descending degree using Algorithm 1, as this ensures that the current pipe is always 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 [13, 45] 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.

4.1 Evaluation Set-Up

We compare our implementation of Synchronized Planarity with the Clustered Planarity implementations `ILP` by Chimani et al. [17, 19] and `HT` by Gutwenger at al. [39]. Both are written in C++ and are part of the OGDF. The `ILP` implementation by Chimani et al. [17, 19] uses the

ABACUS ILP solver [25] 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 [39, Figure 3] and [32, 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 [39, 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 [39].

We combine the Clustered Planarity datasets that were previously used for evaluations on `HT` and `ILP` to form the set `C-OLD` [17, 19, 39]. We apply the preprocessing rules of Gutwenger at al. [39] to all instances and discard instances that become trivial, non-planar or cluster-connected, since the latter are easy to solve [23, 21]. This leaves 1643 instances; see Table 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 by Fink et al. [29]. The source code of the evaluated version of our implementation and all modifications are available at `github.com/N-Coder/syncplan`,[1] while our dataset is on Zenodo with DOI `10.5281/zenodo.7896021`. Our implementation will also be available as integrated part of the OGDF starting with the next release following version 2023.09.

Details on Dataset Generation

The dataset `C-OLD` is comprised of the datasets `P-Small`, `P-Medium`, `P-Large` by Chimani and Klein [19] together with `PlanarSmallR` (a version of `PlanarSmall` [17] with preprocessing applied), `PlanarMediumR` and `PlanarLargeR` by Gutwenger et al. [39]. The preprocessing reduced the dataset of Chimani and Klein [19] to 64 non-trivial instances, leading to dataset `C-OLD` containing 1643 instances in total.

The OGDF library can generate an entirely random (and thus usually non-cluster-planar) clustering by selecting random subsets of vertices. It can also generate a random cluster-connected clustering on a given graph by running a breadth-first search that is stopped at random vertices,

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

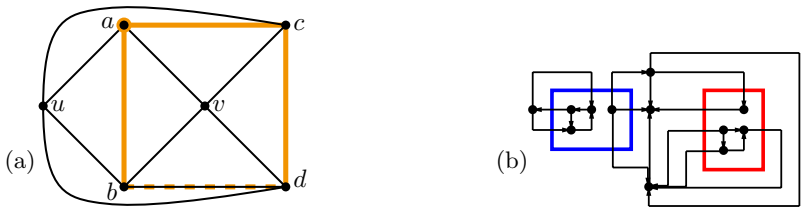


Figure 7: **(a)** Converting the subtree $\{a, b, c, d\}$ with root a (shown in orange) into a cluster will separate vertices u and v, as the edge bd (dashed) will also be part of the cluster. **(b)** A clustered-planar graph with two clusters (in addition to the root cluster) that `HT` classifies as "nonCPlanarVerified".

forming new clusters out of the discovered trees. To also generate non-cluster-connected instances using the same approach, 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 still roughly 90 % of these instances are not clustered-planar (see Table 2). This is because the random BFS-subtree used to generate these clusters only ensures that the generated cluster itself, but not its complement are connected. Thus, if the subgraph induced by the selected vertices contains a cycle, this cycle may separate the outside of the cluster; see Figure 7a. To reliably generate yes-instances, we implemented a third method for generating random clusterings as follows. 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`.

4.2 Results

Table 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

	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 2: Counts of the results 'yes', 'no', 'error', and 'timed out' on C-OLD, C-NCP and C-MED.

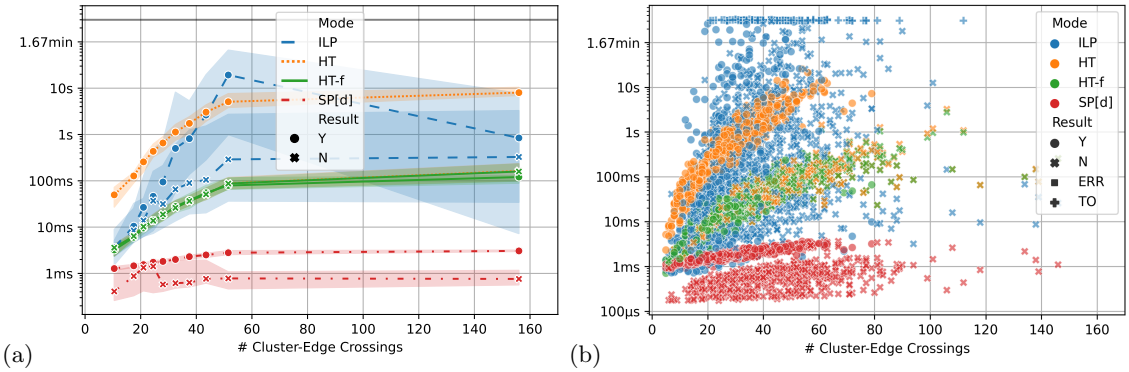


Figure 8: Median running times on dataset C-OLD **(a)** together with the underlying scatter plot **(b)**. For each algorithm, we show running times for yes- and no-instances separately. Markers show medians of bins each containing 10 % of the instances. Shaded regions around each line show inter-quartile ranges.

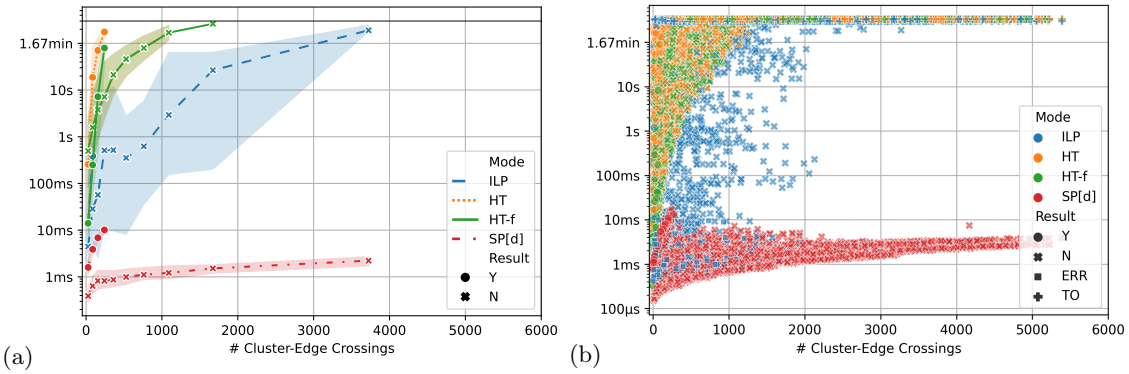


Figure 9: Median running times **(a)** and scatter plot **(b)** on dataset C-NCP.

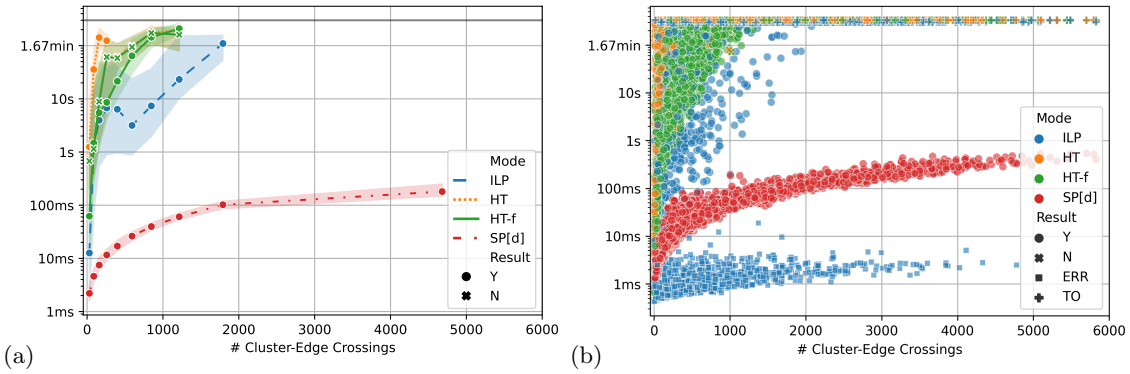


Figure 10: Median running times **(a)** and scatter plot **(b)** on dataset `C-MED`.

ranges in Figure 8 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. The result counts in Table 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 8. `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, often reporting an "undefined optimization result for c-planarity computation" as error; see Table 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. [39] should answer "no" only if the instance truly is negative. Figure 7b shows a minimal minor of the instance for which the results still disagree. We suspect that the issues with both `ILP` and `HT` are implementation bugs and not a conceptual issue of the underlying approaches, although we were not able to easily find fixes for either bug.

The running times on dataset `C-MED` with only positive instances shown in Figure 10 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 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.

5 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 5.1 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 study the effect of these two decisions in Section 5.2. We investigate an alternative method for computing embedding trees in Section 5.3, 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. This new approach is combined with the previously-considered degrees of freedom in Section 5.4. While each subsection ends with its own short summary, we also restate our main take-aways in Section 5.5.

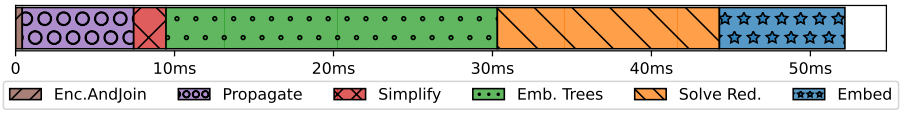


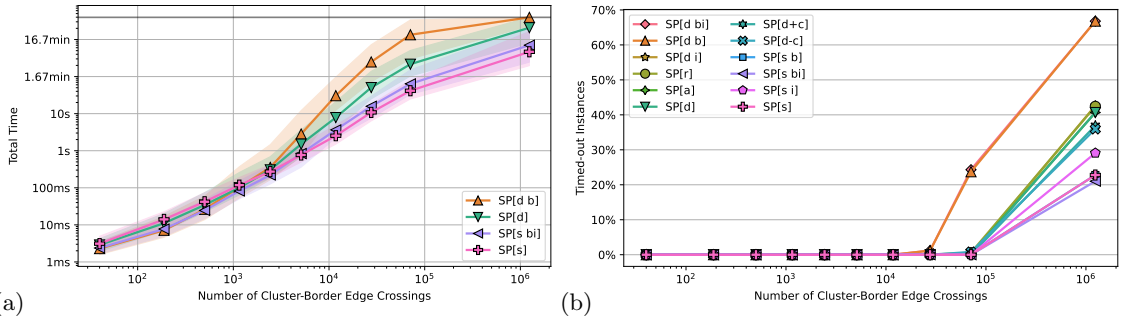
Figure 11: Average time spent on different operations for `SP[d]` on `C-MED`.

Initial Considerations

To gain a first overview over which parts could benefit the most from improvements, Figure 11 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 $n \in \{100, 500, 1000, 5000, 10\,000, 50\,000, 100\,000\}$, $d \in \{1, 10, 100\}$, $c \in \{3, 5, 10, 25, 50, 100, 1000\}$



(a) (b)

Figure 12: `C-LRG` median absolute running times **(a)** and fraction of timeouts **(b)**. Each marker again corresponds to a bin containing 10 % of the instances.

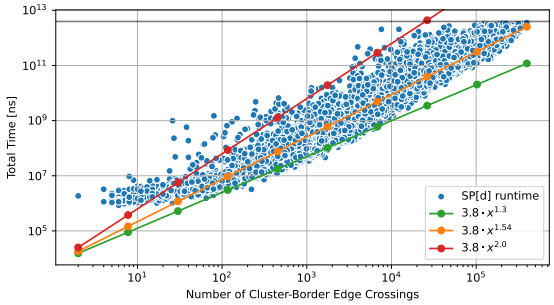
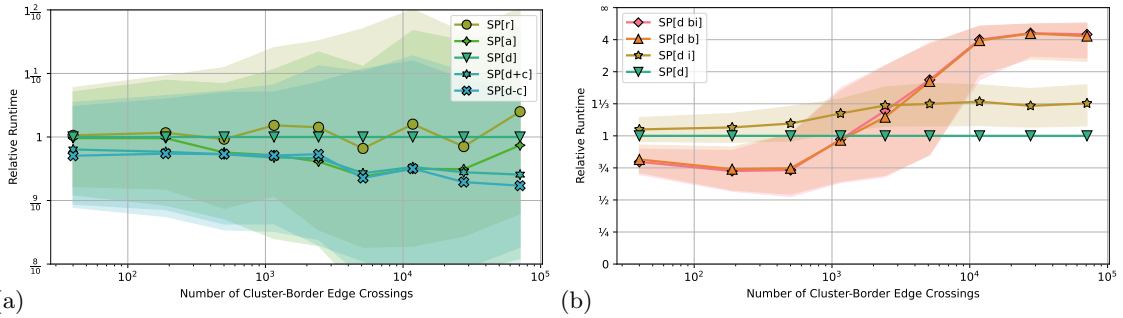


Figure 13: Scatterplot and estimate for `SP[d]` running time growth behavior on `C-LRG`.



(a) (b)

Figure 14: 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.

for dataset `C-LRG`. This yields 6615 instances, out of which 5096 are non-trivial after preprocessing; see Table 1. For the test runs on these large instances, we increase the timeout to 1 hour.

Figure 12a shows 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 12b shows the fraction of runs that timed out for each variant. To get a rough estimate of the practical runtime growth behavior, we fit a polynomial to the runtime data shown in Figure 13 and thereby find the running time growth behavior to be similar to $d^{1.5}$, where d is the number of crossings between edges and cluster borders.

5.1 Pipe Ordering

Recall that, to be able to deliberately choose the next pipe, we use a priority queue of all pipes in the current instance in Algorithm 1, 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 Figure 6). 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 14a 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 12b. Figure 14a 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. 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 6.2. We keep `SP[d]` as the baseline for our further analysis.

5.2 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 14b 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 12a. This is probably caused by the larger connected components (see the last column of Table 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 14b 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. The running times of `SP[d bi]` being very similar to those of `SP[d b]` can be explained by most pipes between block vertices having their endpoints in different connected components. Thus, most of them can directly be joined instead of using a `PropagatePQ` that would result in embedding trees being intersected.

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.

Algorithm 2: Modified algorithm for removing all pipes using SPQR-trees.

```
while !pipes.empty() do
    while !cut-cut-pipes.empty() do
        EncapsulateAndJoin (cut-cut-pipes.pop());
    B ← biconnected component of one endpoint of pipes.top();
    S ← SPQR-tree of B;
    for block-block pipe ρ with endpoint x in B do
        T ← embedding tree of x derived from S;
        apply remainder of Algorithm 1 starting with the if-condition of Line 2;
```

5.3 Batched Embedding Tree Generation

Our preliminary analysis showed that the computation of embedding trees consumes a large fraction of the running time (see Figure 11), 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

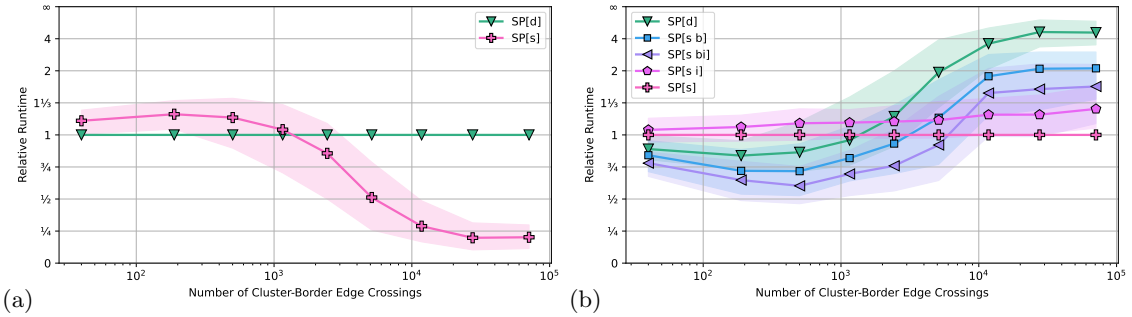


Figure 15: Relative running times for **(a)** SPQR-tree batched embedding tree generation and **(b)** for different variants thereof.

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. [12]: Each occurrence of v in a parallel skeleton of the SPQR-tree corresponds to a (PQ-tree) P-node in the embedding tree of v, each occurrence in a rigid skeleton to a (PQ-tree) Q-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.

Our modified procedure for removing all pipes using this batched embedding tree computation based on SPQR-trees is shown in Algorithm 2. Note that we can also derive the embedding tree T' used in Line 5 of the re-used Algorithm 1 from $\mathcal{S}$.

We use Algorithm 2 in variant `SP[s]`. Figures 12a and 15a 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.

5.4 SPQR-Batch Variations

Figure 15b switches the baseline between the two variants shown in Figure 15a and additionally contains combinations of the variants from Section 5.2 with the SPQR-batch computation. As in Figure 14b, 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 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 12a, where `SP[s]` is more than an order of magnitude faster than `SP[d b]` on large instances.

5.5 Summary

In this section, we studied how degrees of freedom in the Synchronized Planarity algorithm can be used to improve the running times on yes-instances. We investigated deliberately choosing the order in which pipes are processed (Section 5.1), handling block-block pipes through direct joins or by intersecting their embedding trees (Section 5.2), and combining these with a more costly SPQR-tree-based approach that yields multiple embedding trees at once (Sections 5.3 and 5.4). Regarding the first point, we found no major advantage in choosing pipes in a certain order. While joining block-block pipes yields a small advantage on small instances, this turns into a strong disadvantage for larger instances. This is probably due to connected components growing larger through the joins, thus making the embedding tree calculation even more costly. Similarly, computing and intersecting embedding trees for both endpoints of a pipe noticeably increases the overhead spent on their generation, while also not making sufficiently more progress to compensate for this.

The biggest improvement can be seen by using SPQR-trees to compute multiple embedding trees at once. While the more costly computation is still noticeable for small instances, this is quickly outweighed for larger instances as components have to be processed in their entirety fewer times. For small instances, we found that the additional overhead can be more than compensated by using the speed-up of joining block-block pipes only on these instances. In conclusion, for Synchronized Planarity instances derived from Clustered Planarity, the ordering of pipes has no relevant effect, embedding trees should clearly always be computed in batches via SPQR-trees, while block-block pipes can optionally be handled specially on sufficiently small instances.

6 Further Analysis

To strengthen our conclusions drawn from the results in the previous section, we will in this section provide further in-depth analysis of the runtime behavior of the different variants. Furthermore, we analyze their performance on three further datasets generated from entirely different problems to give a more general 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 6.1, where Figure 16 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 section, we also use a statistical test to check whether one variant is significantly faster than another. The results presented in Section 6.2 underline 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 three further datasets `SEFE-LRG`, `SP-LRG`, and `LVL-LRG` are presented in Section 6.3 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`].

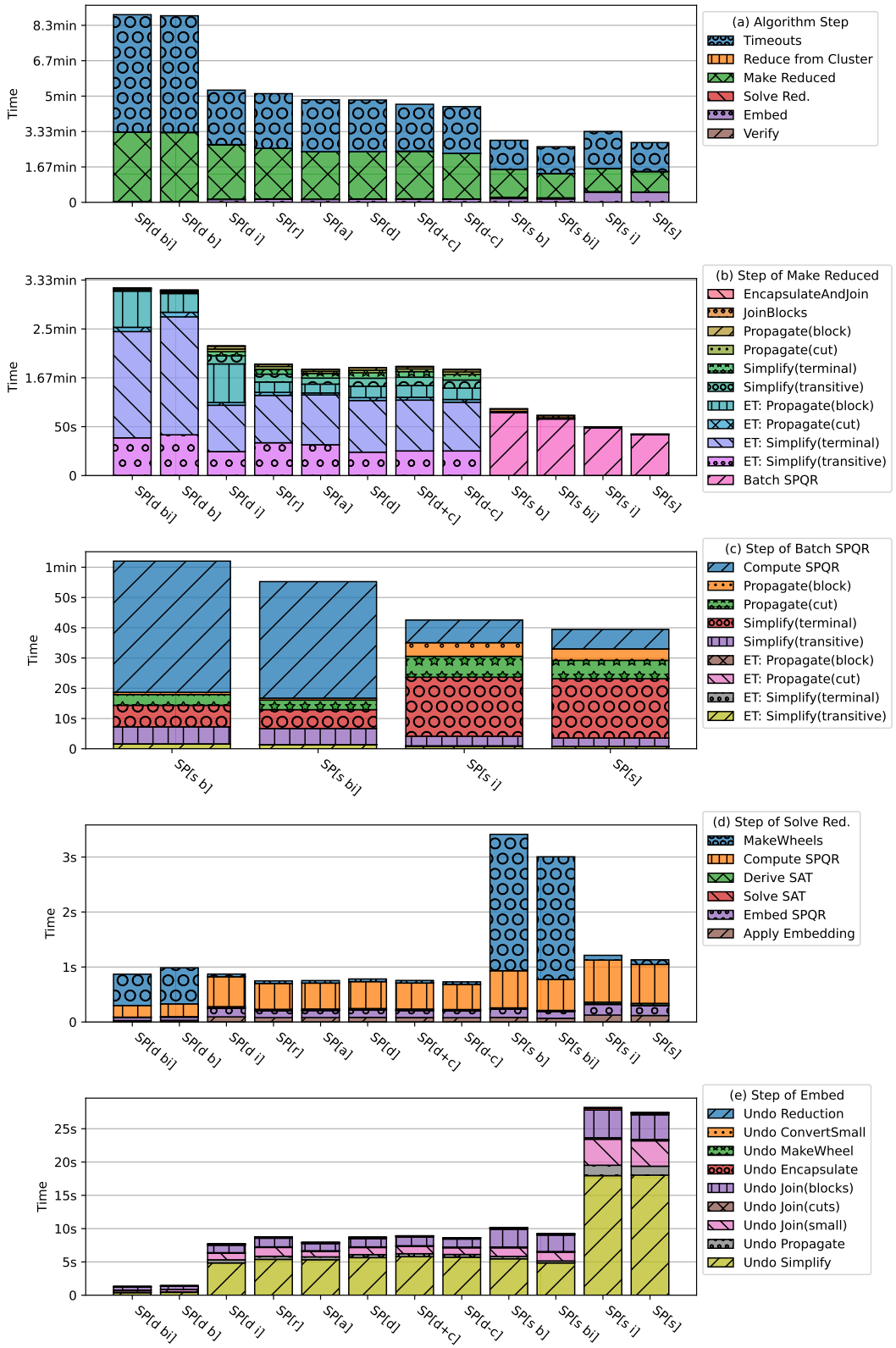


Figure 16: The average running time of our different Synchronized Planarity variants.

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 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.

6.1 Detailed Runtime Profiling

Table 3 shows the per-step running time information aggregated for variants studied in the previous section. Figure 16 in greater detail shows how the running time spent is split on average across the different steps of the algorithm (Figure 16a) and then also further drills down on the composition of the individual steps that make the instance reduced (Figure 16b), solve the reduced instance (Figure 16d), and then derive a solution and an embedding for the input instance by undoing all changes while maintaining the embedding (Figure 16e). 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 16c). 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 12a.

Figure 16b 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 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 16c. Here, the terminal case of the `SimplifyMatching` operation (described in the bottom left part of Figure 5) 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 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 16d. 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 16e 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 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 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 even more so the processing order of pipes do not seem to have a large 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`].

6.2 Statistical Significance

To test whether one variant is (in the statistical sense) significantly faster than another, we use the methodology proposed by Radermacher [47, 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 [47], 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 $\mathcal{G}_{\text{train}}$ 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.

Applying this methodology, Figure 17 compares the pairwise advantages of the variants from

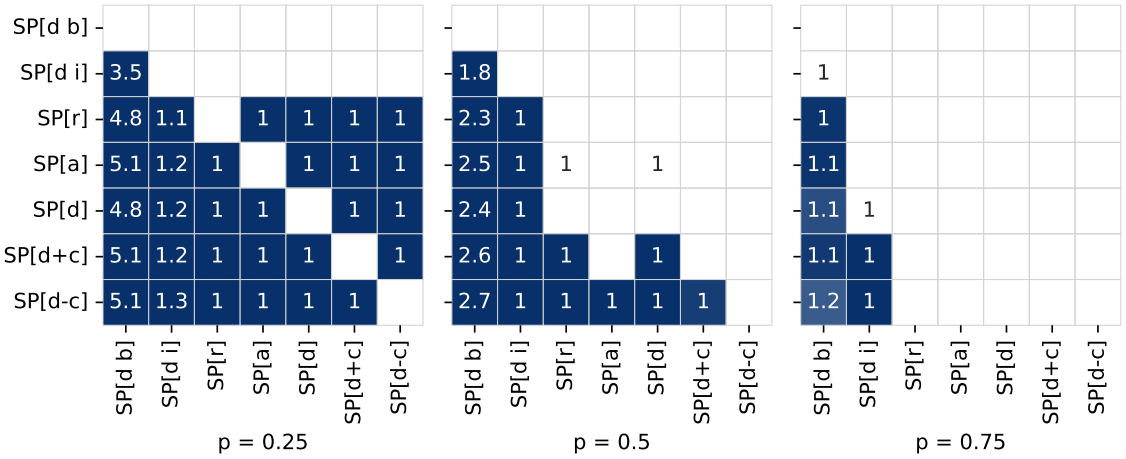


Figure 17: Advantages of variants without flag [s] on C-LRG instances of size at least 5000. Blue cell backgrounds indicate significant values, while in cells with white background, we were not able to reject the null-hypothesis with significance $\alpha = 0.05$. Empty cells indicate that the fraction where one algorithm is better than the other is smaller than p.

Sections 5.1 and 5.2. 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 14a. For half of the dataset, SP[d-c] is still slightly faster than other variants, while no variant from Section 5.1 is faster than another for at least three quarters of instances. To summarize, our results here corroborate the findings from Sections 5.1 and 5.2, 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.

Figures 18a and 18b compare the pairwise advantages of the variants from Sections 5.3 and 5.4 (see also Figure 15b) for instances with more and less than 5000 cluster-border edge crossings, respectively. For the larger instances of Figure 18a, 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 18b, 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 5.3 and 5.4, where for large instances variant SP[s] is the fastest, whereas for smaller instances SP[s bi] is superior.

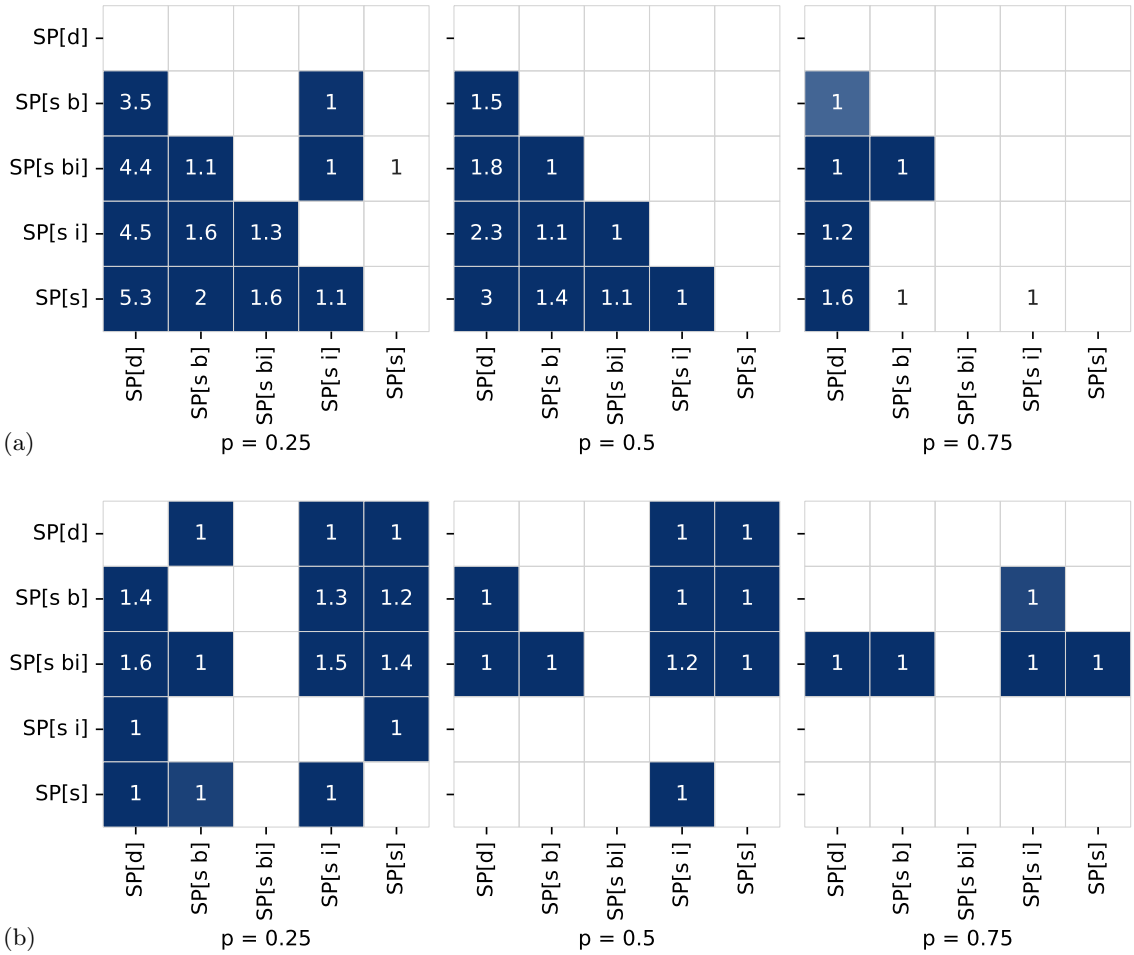


Figure 18: Advantages of variants with flag [s] on C-LRG instances of size at least 5000 (a) and at most (b) 5000.

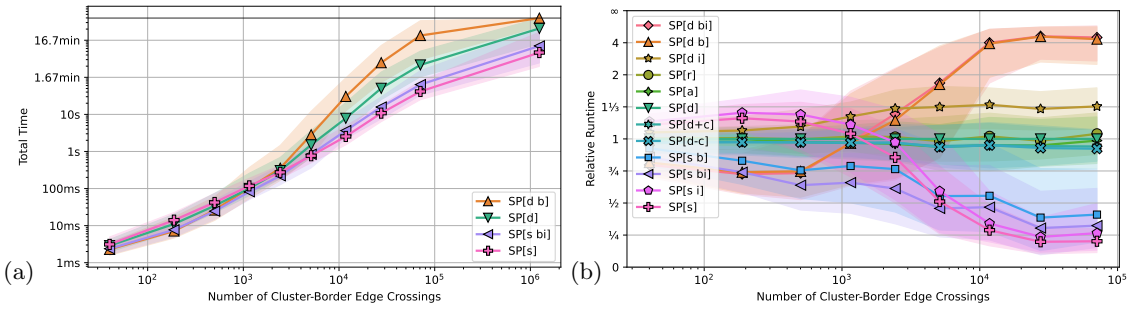


Figure 19: Absolute (a) and relative (b) running times with regard to `SP[d]` for `C-LRG`.

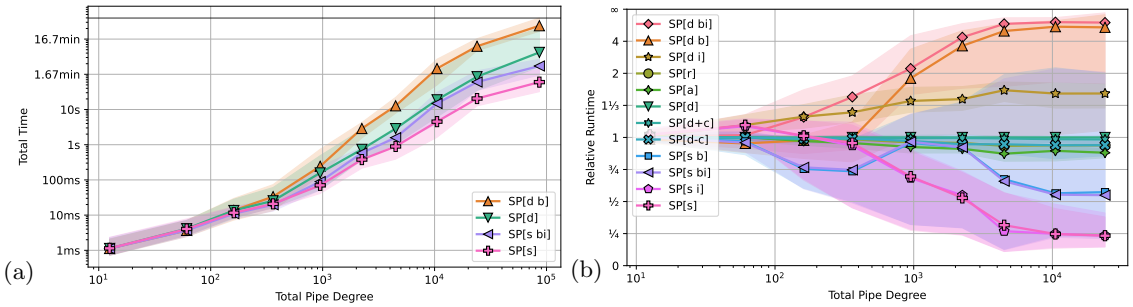


Figure 20: Absolute (a) and relative (b) running times with regard to `SP[d]` for `SP-LRG`.

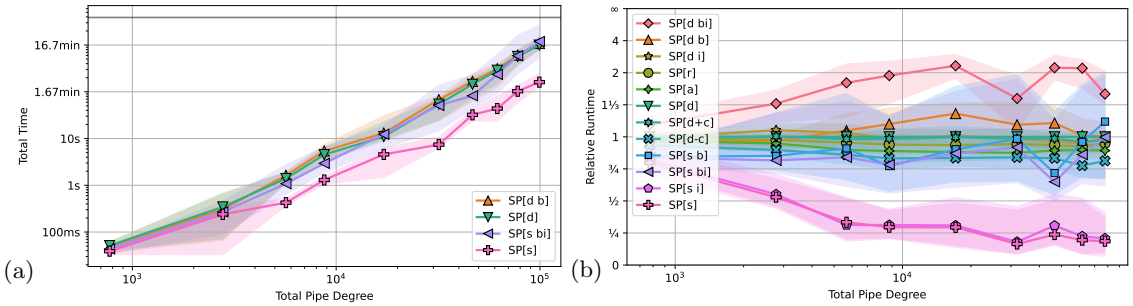


Figure 21: Absolute (a) and relative (b) running times with regard to `SP[d]` for `SEFE-LRG`.

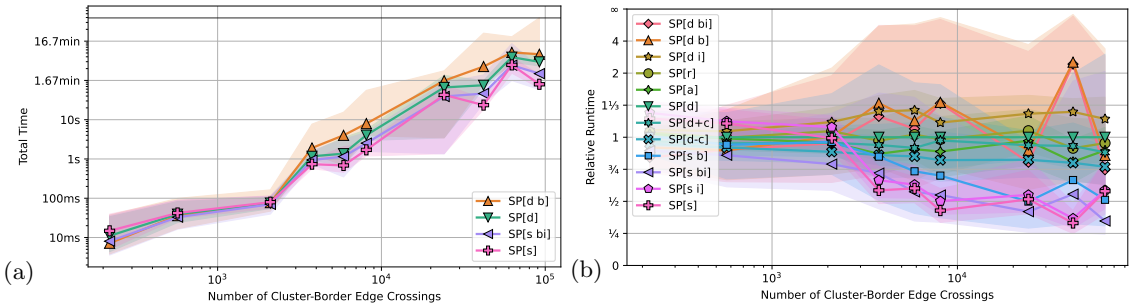


Figure 22: Absolute (a) and relative (b) running times with regard to `SP[d]` for `LVL-LRG`.

6.3 Other Problem Instances

In addition to the Clustered Planarity dataset used up to now we also generate three further datasets from different problems. The first dataset uses the reduction from Connected SEFE indicated in Figure 3. 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 $b \in \{0.25, 0.5, 0.75, 1\}$, we then add $(3n - 6 - m) \cdot b$ edges to each exclusive graph, i.e., the fraction b 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 b. This leads to the dataset `SEFE-LRG` containing 1008 instances; see Table 1.

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 $c \in \{1, 10, 100\}$ distinct connected components. Note that we do not include graphs that would have no edges, e.g., those with $n = 100$ and $c = 100$. 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; see Table 1.

Finally, we generate a dataset derived from proper (Radial) Level Planarity instances. To do so, we start with a random, maximal embedded Level Planarity instance (optionally also adding edges between the first and last vertices of adjacent levels in the radial case), from which we delete edges until we reach the desired density. We generate six instances (three radial and three plane) for each combination of $n \in \{100, 500, 1000, 5000, 10\,000, 50\,000, 100\,000\}$ nodes, $m \in \{n, 1.25n, 1.5n, 1.75n, 2n\}$ edges, and $\ell \in \{5, 10, 25, 50, 100, 250, 500, 1000\}$ levels. Note that we do not include instances with $\frac{n}{4} < \ell$, i.e., those with on average less than four nodes per level. We then use a reduction that turns levels into (concentric nested) clusters such that the number of cluster-border edge crossings stays linear in the number of edges [28, Section 6.4.4]. This leads to the dataset `LVL-LRG` that contains 1103 Clustered Planarity instances after preprocessing. Altogether, our seven datasets contain 29 442 instances in total; see Table 1.

Results

Running the same evaluation on the datasets `SEFE-LRG`, `SP-LRG`, and `LVL-LRG` yielded absolute running times with roughly the same orders of magnitude as for `C-LRG`, see the left plots in Figures 19 to 22 (but note that the plots show slightly different ranges on both axes). The right plots in the figures again detail the running times relative to `SP[d]`. For `SP-LRG`, the relative running time behavior is similar to the behavior observed on `C-LRG`. The two major differences concern variants with flag [b]. Variants `SP[d b(i)]` (that is both `SP[d b]` and `SP[d bi]`) are not faster than `SP[d]` on small instances and also sooner grow slower on large instances. Similarly, `SP[s b(i)]` is not much faster than `SP[d]` on small instances, and the 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 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 smaller than in the two previous datasets, and especially both `SP[d b]` and `SP[d i]` show no major slowdown over `SP[d]`. The difference between `SP[d b]` and `SP[d bi]` thus being more pronounced could indicate more block-block pipes with endpoints in the same connected components than in the previous datasets, leading to more costly intersections instead of joins being performed. In the other direction, `SP[s b(i)]` is at no point faster than `SP[s (i)]`, even for small instances, and even has no noticeable advantage over `SP[d]`. Interestingly, in comparison the the two previous datasets, joining block-block pipes thus now completely removes the advantage of the SPQR-tree-based embedding tree computation, while it also seems to no longer have a major disadvantage without it. As variants `SP[s (i)]` are faster than both `SP[s b(i)]` and `SP[d]` even for small instances, they are the fastest for all instances sizes.

The instances distribution varies even more for `LVL-LRG`, where the total pipe degree (or equivalently, the number of cluster-border edge crossings) directly corresponds to the number of edges in the (Radial) Level Planarity instance. While `SP[s]` and its variations are still clearly the fastest on large instances, the median slow-down through the optional join of block-block pipes is far less pronounced both for `SP[s b(i)]` and `SP[d b(i)]` here. Note that the interquartile range shown in Figure 22b still indicates a large possible slow-down for `SP[d b(i)]`, with a large variance in individual running times.

To summarize, running our algorithm on these three additional datasets still yields results that mostly agree with the results on `C-LRG`, with `SP[d b(i)]` among 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], and thus the point at which `SP[s (i)]` starts being faster than `SP[s b(i)]` (see also Section 5.4).

6.4 Summary

In this section, we use three different approaches to strengthen the insights and conclusions which we summarized at the end of the previous section in Section 5.5. First, still using the same `C-LRG` dataset, we analyzed how the overall running times are split across the different parts of the algorithm, and also linked this to statistics of the instance processed by our algorithm (Section 6.1). Second, we used a statistical test to validate the significance of the observed advantages of different variants in this dataset (Section 6.2). Third, we generated three further datasets, using reductions from further problems and thus also very different instance generators, and thereby verified how our insights transfer to other classes of instances (Section 6.3).

The runtime profiling information underlines that computing embedding trees takes the by far largest fraction of running time for almost all variants, while the batched SPQR-tree approach is able to save big parts of that time. Joining block-block pipes increases the maximum size of biconnected components by close to an order of magnitude, and thus makes the embedding tree computation take even longer. Still, joining does preempt many pipes that need to be removed via `SimplifyMatching`, although this advantage of needing fewer operations is only noticeable for the fastest variants.

Out statistical analysis again shows that changing pipe ordering yields no significant advantages, while joining block-block pipes or intersecting embedding trees can cause significant slowdowns when computing the trees individually. In contrast, deriving embedding trees in batches from a SPQR-tree does yield a significant speed-up for large instances. For smaller instances, this is only true for `SP[s bi]`, i.e., the combination of all variations yields the greatest advantages.

Running our algorithm on three further datasets from entirely different problems and instance generators still yields results that mostly agree with the results on `C-LRG`, although the magnitude of the overhead generated by large connected components varies for variants with flag [`b`]. Due to this, among our fastest variants, the point at which `SP[s (i)]` starts being faster than `SP[s b(i)]` changes depending on the class of instances.

7 Conclusion

In this paper, we described the first practical implementation of Synchronized Planarity, which generalizes many constrained planarity problems such as Clustered Planarity and Connected SEFE. We evaluated close to 30 000 instances stemming from different problems. Using the quadratic algorithm by Bläsius et al. [8], instances with 100 vertices are solved in milliseconds, while we can still 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 by Fink and Rutter [30], 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 [16].

References

[1] P. Angelini and G. Da Lozzo. SEFE = C-Planarity? *The Computer Journal*, 59(12):1831–1838, 2016. doi:10.1093/comjnl/bxw035.

[2] P. Angelini and G. Da Lozzo. Clustered planarity with pipes. *Algorithmica*, 81(6):2484–2526, 2019. doi:10.1007/s00453-018-00541-w.

[3] P. Angelini, G. Da Lozzo, and D. Neuwirth. Advancements on SEFE and partitioned book embedding problems. *Theoretical Computer Science*, 575:71–89, 2015. doi:10.1016/j.tcs.2014.11.016.

[4] P. Angelini, M. Di Bartolomeo, and G. Di Battista. Implementing a partitioned 2-page book embedding testing algorithm. In W. Didimo and M. Patrignani, editors, *Proceedings of the 20th International Symposium on Graph Drawing (GD'12)*, volume 7704 of *Lecture Notes in Computer Science*, pages 79–89. Springer, 2012. doi:10.1007/978-3-642-36763-2_8.

[5] P. Angelini, G. Di Battista, F. Frati, V. Jelínek, J. Kratochvíl, M. Patrignani, and I. Rutter. Testing planarity of partially embedded graphs. *ACM Transactions on Algorithms*, 11(4):32:1–32:42, 2015. doi:10.1145/2629341.

[6] P. Angelini, G. Di Battista, F. Frati, M. Patrignani, and I. Rutter. Testing the simultaneous embeddability of two graphs whose intersection is a biconnected or a connected graph. *Journal of Discrete Algorithms*, 14:150–172, 2012. doi:10.1016/j.jda.2011.12.015.

[7] C. Bachmaier. *Circle planarity of level graphs*. PhD thesis, University of Passau, Germany, 2004. URL: https://nbn-resolving.org/urn:nbn:de:bvb:739-opus-385.

[8] T. Bläsius, S. D. Fink, and I. Rutter. Synchronized planarity with applications to constrained planarity problems. *ACM Transactions on Algorithms*, 19(4), 2023. doi:10.1145/3607474.

[9] T. Bläsius, S. G. Kobourov, and I. Rutter. Simultaneous embedding of planar graphs. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 11, pages 349–381. 2013. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/simultaneous.pdf.

[10] T. Bläsius and I. Rutter. Disconnectivity and relative positions in simultaneous embeddings. *Computational Geometry. Theory and Applications*, 48(6):459–478, 2015. doi:10.1016/j.comgeo.2015.02.002.

[11] T. Bläsius and I. Rutter. A new perspective on clustered planarity as a combinatorial embedding problem. *Theoretical Computer Science*, 609:306–315, 2016. arXiv:1506.05673, doi:10.1016/j.tcs.2015.10.011.

[12] T. Bläsius and I. Rutter. Simultaneous PQ-ordering with applications to constrained embedding problems. *ACM Transactions on Algorithms*, 12(2):16:1–16:46, 2016. doi:10.1145/2738054.

[13] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976. doi:10.1016/S0022-0000(76)80045-1.

[14] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. 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)*, pages 25–36. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-24595-7_3.

[15] P. Braß, E. Cenek, C. A. Duncan, A. Efrat, C. Erten, D. Ismailescu, S. G. Kobourov, A. Lubiw, and J. S. B. Mitchell. On simultaneous planar graph embeddings. *Computational Geometry. Theory and Applications*, 36(2):117–130, 2007. doi:10.1016/j.comgeo.2006.05.006.

[16] G. Brückner. *Planarity Variants for Directed Graphs*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2021. URL: https://nbn-resolving.org/urn:nbn:de:101:1-20210804050229888868936.

[17] M. Chimani, C. Gutwenger, M. Jansen, K. Klein, and P. Mutzel. Computing maximum c-planar subgraphs. In I. G. Tollis and M. Patrignani, editors, *Proceedings of the 16th International Symposium on Graph Drawing (GD'08)*, volume 5417 of *Lecture Notes in Computer Science*, pages 114–120. Springer, 2008. doi:10.1007/978-3-642-00219-9_12.

[18] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. The Open Graph Drawing Framework (OGDF). In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 17, pages 543–569. 2014. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/ogdf.pdf.

[19] M. Chimani and K. Klein. Shrinking the search space for clustered planarity. In W. Didimo and M. Patrignani, editors, *Proceedings of the 20th International Symposium on Graph Drawing (GD'12)*, volume 7704 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2012. doi:10.1007/978-3-642-36763-2_9.

[20] S. Cornelsen and D. Wagner. Completely connected clustered graphs. *Journal of Discrete Algorithms*, 4(2):313–323, 2006. doi:10.1016/J.JDA.2005.06.002.

[21] P. F. Cortese, G. Di Battista, F. Frati, M. Patrignani, and M. Pizzonia. C-planarity of c-connected clustered graphs. *Journal of Graph Algorithms and Applications*, 12(2):225–262, 2008. doi:10.7155/jgaa.00165.

[22] G. 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.

[23] E. Dahlhaus. A linear time algorithm to recognize clustered planar graphs and its parallelization. In C. L. Lucchesi and A. V. Moura, editors, *Proceedings of the 3rd Latin American Symposium (LATIN'98)*, pages 239–248. Springer Berlin Heidelberg, 1998. doi:10.1007/bfb0054325.

[24] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, 1996. doi:10.1007/bf01961541.

[25] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi. Branch-and-cut algorithms for combinatorial optimization and their implementation in abacus. In M. Jünger and D. Naddef, editors, *Computational Combinatorial Optimization: Optimal or Provably Near-Optimal Solutions*, pages 157–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi:10.1007/3-540-45586-8_5.

[26] A. Estrella-Balderrama, J. J. Fowler, and S. G. Kobourov. Graphset, a tool for simultaneous graph drawing. *Software: Practice and Experience*, 40(10):849–863, 2010. doi:10.1002/spe.958.

[27] Q.-W. Feng, R. F. Cohen, and P. Eades. Planarity for clustered graphs. In P. G. Spirakis, editor, *Proceedings of the 3rd Annual European Symposium on Algorithms (ESA'95)*, volume 979 of *LNCS*, pages 213–226. Springer, 1995. doi:10.1007/3-540-60313-1_145.

[28] S. D. Fink. *Constrained Planarity Algorithms in Theory and Practice*. PhD thesis, Universität Passau, 2024. doi:10.15475/cpatp.2024.

[29] S. D. Fink, M. Pfretzschner, and I. Rutter. Experimental comparison of pc-trees and pq-trees. *ACM Journal of Experimental Algorithmics*, 28, 2023. doi:10.1145/3611653.

[30] S. D. Fink and I. Rutter. Maintaining triconnected components under node expansion. *Computing in Geometry and Topology*, pages 202–216, 2023. doi:10.1007/978-3-031-30448-4_15.

[31] H. D. Fraysseix, P. O. D. Mendez, and P. Rosenstiehl. Trémaux trees and planarity. *International Journal of Foundations of Computer Science*, 17(05):1017–1029, 2006. doi:10.1142/S0129054106004248.

[32] R. Fulek, J. Kynčl, I. Malinović, and D. Pálvölgyi. Clustered planarity testing revisited. *The Electronic Journal of Combinatorics*, 22(4), 2015. doi:10.37236/5002.

[33] R. Fulek, M. J. Pelsmajer, M. Schaefer, and D. Štefankovič. Hanani–tutte, monotone drawings, and level-planarity. In *Thirty Essays on Geometric Graph Theory*, pages 263–287. Springer New York, 2012. doi:10.1007/978-1-4614-0110-0_14.

[34] R. Fulek and C. D. Tóth. Atomic embeddability, clustered planarity, and thickenability. *Journal of the ACM*, 69(2):13:1–13:34, 2022. arXiv:1907.13086v1, doi:10.1145/3502264.

[35] E. Gassner, M. Jünger, M. Percan, M. Schaefer, and M. Schulz. Simultaneous graph embeddings with fixed edges. In F. V. Fomin, editor, *Proceedings of the 32nd Workshop on Graph-Theoretic Concepts in Computer Science (WG'06)*, volume 4271 of *Lecture Notes in Computer Science*, pages 325–335. Springer, 2006. doi:10.1007/11917496_29.

[36] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988. doi:10.1145/48014.61051.

[37] C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan, and R. Weiskircher. Advances in c-planarity testing of clustered graphs. In S. G. Kobourov and M. T. Goodrich, editors, *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 220–235. Springer, 2002. doi:10.1007/3-540-36151-0_21.

[38] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In J. Marks, editor, *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of *LNCS*, pages 77–90. Springer, 2000. doi:10.1007/3-540-44541-2_8.

[39] C. Gutwenger, P. Mutzel, and M. Schaefer. Practical experience with hanani-tutte for testing c-planarity. In C. C. McGeoch and U. Meyer, editors, *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments (ALENEX'14)*, pages 86–97. Society for Industrial and Applied Mathematics, 2014. doi:10.1137/1.9781611973198.9.

[40] M. Harrigan and P. Healy. Practical level planarity testing and layout with embedding constraints. In S. Hong, T. Nishizeki, and W. Quan, editors, *Proceedings of the 15th International Symposium on Graph Drawing (GD'07)*, volume 4875 of *Lecture Notes in Computer Science*, pages 62–68. Springer, 2007. doi:10.1007/978-3-540-77537-9_9.

[41] S.-H. Hong and H. Nagamochi. Two-page book embedding and clustered graph planarity. Technical Report TR[2009-004], 2009. URL: https://citeseerx.ist.psu.edu/doc/10.1.1.361.1233.

[42] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. In S. Whitesides, editor, *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 224–237. Springer, 1998. doi:10.1007/3-540-37623-2_17.

[43] S. Leipert. *Level planarity testing and embedding in linear time*. PhD thesis, Universität zu Köln, 1998.

[44] T. Lengauer. Hierarchical planarity testing algorithms. *Journal of the ACM*, 36(3):474–509, 1989. doi:10.1145/65950.65952.

[45] M. Patrignani. Planarity testing and embedding. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 1, pages 1–42. 2013. URL: https://cs.brown.edu/people/rtamassi/gdhandbook/chapters/planarity.pdf.

[46] H. C. Purchase, J.-A. Allder, and D. Carrington. Graph layout aesthetics in UML diagrams: User preferences. *Journal of Graph Algorithms and Applications*, 6(3):255–279, 2002. doi:10.7155/jgaa.00054.

[47] M. Radermacher. *Geometric Graph Drawing Algorithms - Theory, Engineering and Experiments*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2020. doi:10.5445/IR/1000117664.

[48] B. Randerath, E. Speckenmeyer, E. Boros, P. L. Hammer, A. Kogan, K. Makino, B. Simeone, and O. Cepek. A satisfiability formulation of problems on level graphs. *Electronic Notes in Discrete Mathematics*, 9:269–277, 2001. doi:10.1016/S1571-0653(04)00327-0.

[49] I. Rutter. Simultaneous embedding. In S.-H. Hong and T. Tokuyama, editors, *Beyond Planar Graphs*, chapter 13, pages 237–265. Springer Singapore, 2020. doi:10.1007/978-981-15-6533-5_13.

[50] M. Schaefer. Toward a theory of planarity: Hanani-tutte and planarity variants. *Journal of Graph Algorithms and Applications*, 17(4):367–440, 2013. doi:10.7155/jgaa.00298.

[51] C. Ware, H. Purchase, L. Colpoys, and M. McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, June 2002. doi:10.1057/palgrave.ivs.9500013.