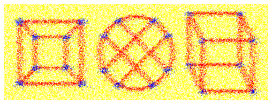
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 30, no. 2, pp. 119–145 (2026)
DOI: 10.7155/jgaa.v30i2.3136

Heuristics for Exact 1-Planarity Testing

Simon D. Fink[1] *Miriam Münch*[2] *Matthias Pfretzschner*[2] *Ignaz Rutter*[2]

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

Submitted: Nov. 2025 Accepted: April 2026 Published: May 2026

Article type: Regular paper Communicated by: V. Dujmović, F. Montecchiani

Abstract. Since many real-world graphs are nonplanar, the study of graphs that allow few crossings per edge has been an active subfield of graph theory in recent years. One of the most natural generalizations of planar graphs are the so-called 1-planar graphs that admit a drawing with at most one crossing per edge. Unfortunately, testing whether a graph is 1-planar is known to be NP-complete even for very restricted graph classes. On the positive side, Binucci, Didimo and Montecchiani [7] presented the first practical algorithm for testing 1-planarity based on an easy-to-implement backtracking strategy. We build on this idea and systematically explore the design choices of such algorithms and propose several new ingredients, such as different branching strategies and multiple filter criteria that allow us to reject certain branches in the search tree early on. We conduct an extensive experimental evaluation that evaluates the efficiency and effectiveness of these ingredients. Given a time limit of three hours per instance, our best configuration is able to solve more than 98% of the non-planar instances from the well-known `North` and `Rome` graphs with up to 50 vertices. Notably, the median running time for solved instances is well below 1 second. Our backtracking framework can also be used for testing IC- and NIC-planarity.

1 Introduction

It has been established early on in the history of Graph Drawing that the number of crossings has a major influence on the readability of drawings [43, 44, 45]. This reinforces the continued interest in the extensively studied *crossing minimization problem*, which asks for the minimum

Special issue on Selected papers from the Thirty-third International Symposium on Graph Drawing and Network Visualization, GD 2025

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 541433306. Simon D. Fink acknowledges support from Projects No. 10.47379/ICT22029 of the Vienna Science Foundation (WWTF) and No. 10.55776/Y1329 of the Austrian Science Fund (FWF).

E-mail addresses: sfink@ac.tuwien.ac.at (Simon D. Fink) muenchm@fim.uni-passau.de (Miriam Münch) pfretzschner@fim.uni-passau.de (Matthias Pfretzschner) rutter@fim.uni-passau.de (Ignaz Rutter)


This work is licensed under the terms of the CC-BY license.

number of crossings over all drawings of a given graph. It was shown as early as 1983 that the problem is NP-complete [24]. Due to its foundational nature, it has fueled a large number of results that address combinatorial and algorithmic aspects of crossing numbers. We refer to the survey of Schaefer for an overview of this extensive field [50]. Due to the practical importance of the problem, these developments have early on been paralleled by more applied works that are concerned with solving the crossing minimization problem in practice. Today, there is a large amount of literature that deals with computing the crossing number either exactly in exponential time, based on formulations as integer linear programs [11, 16, 38], or heuristically in polynomial time [14, 15, 26]. Heuristics often work by efficiently and optimally inserting vertices or edges into a drawing, allowing to introduce crossings only between pairs of edges where at least one of the edges is currently being inserted; a strategy that has also proved to be successful for straight-line drawings [8, 46, 47, 48].

A more recent development is based on the hypothesis that in addition to the number of crossings, a major influence on the readability of drawings comes from the distribution of the crossings within the drawing. This has lead to the definition of so-called *beyond-planar graphs* that admit drawings whose crossings avoid certain patterns that are considered to impede readability. Prominent examples of beyond-planarity concepts are k-planarity, k-quasi-planarity, fan-planarity, gap-planarity, RAC-planarity, min-k-planarity as well as corresponding outer and upward variants; we refer to [30] for a survey. One of the most widely known beyond-planarity concepts is *1-planarity*, where a graph is called 1-planar if it can be drawn in the plane such that each edge has at most one crossing. Despite the apparent simplicity, 1-planar graphs form a rich class that has spurred a substantial amount of research. From a combinatorial perspective, questions about density, coloring and structural decompositions have been deeply investigated; we refer to [31, 51] for surveys.

From an algorithmic perspective, it is well known that testing 1-planarity is NP-complete [25, 32], even for graphs of bounded bandwidth, pathwidth or treewidth [6], as well as for graphs that can be made planar by the removal of a single edge [12] or graphs with a fixed rotation system [3]. Polynomial-time recognition algorithms are only known for restricted subclasses such as outer 1-planar graphs [2, 29], maximal 1-planar graphs with a fixed rotation system [23], 1-planar maximal graphs of pathwidth w [36], and optimal 1-planar graphs [9]. Recent years have seen a plethora of fixed-parameter tractability (FPT) results on testing 1-planarity. In particular, Bannister, Cabello and Eppstein [6] showed that the problem is FPT with respect to the vertex cover number, the tree-depth, and the cyclomatic number. Various recent results, see e.g. [20, 27, 28, 37], imply that 1-planarity is FPT with respect to the 1-planar crossing number, i.e., the minimum number of crossings over all 1-planar drawings of a graph. The mentioned results are however only of theoretical interest as they rely on deep theoretical results such as Courcelle's Theorem [19] or testing embeddability of graphs in 2-complexes [21].

A key motivation that initiated the study of beyond-planarity was the observation that while many real world-graphs are non-planar, they are usually sparse and close to planarity in the sense described above, namely that they have drawings whose crossings are well-scattered [30]. Correspondingly, the problem of recognizing beyond-planar graphs and in particular testing 1-planarity has a strong practical motivation. Despite this and in stark contrast to the problem of crossing optimization, where the theoretical developments have been accompanied by corresponding practical results, there is a distinct scarcity of practical algorithms in the area of beyond-planarity. In fact, we are only aware of two attempts.

Angelini, Bekos, Kaufmann and Schneck [1] present an algorithm that is based on enumerating all topological drawings of a graph incrementally by iteratively inserting the edges. This is useful,

e.g., to devise small counterexamples or to confirm conjectures on small graphs with less than 15 vertices, but it is hardly useful for practical applications. Binucci, Didimo and Montecchiani [7] present an easy-to-implement algorithm, called `1PlanarTester`, for recognizing 1-planar graphs that is based on backtracking and is quite effective for small graphs. As a graph is 1-planar if and only if all its biconnected components are 1-planar, they process the biconnected components independently. For each biconnected component, they fix a global ordering σ on the set of all pairs of non-adjacent edges, consider each such *crossing candidate* pair $\{e, f\}$ in the order of σ and branch on the binary decision of whether e and f should cross. This is repeated until a 1-planar drawing is found or the whole search tree has been explored and the instance is rejected. Branches that cannot lead to a solution, e.g., when an edge receives more than one crossing, or there is a non-planar subgraph whose edges may not receive further crossings, are excluded from the backtracking. The authors extend this basic algorithm by combining it with several *filter criteria* that exclude additional branches, e.g., based on the addition of "kite" edges around crossings and density bounds for planar and 1-planar graphs. The algorithm can handle graphs with up to 30 vertices in a reasonable amount of time. However, on graphs with up to 50 vertices the algorithm often either gives a positive answer in less than 0.2 seconds or does not give an answer within a three-hour time limit. This emphasizes the crucial role of the order σ; either a lucky guess of helpful crossings early on paves the ways towards a solution, or the algorithm more or less explores the entire search tree.

Contribution and Outline. We build on the ideas of Binucci et al. [7] and systematically explore different design choices within a backtracking framework. To this end we propose new branching strategies aimed at limiting the size of the tree that has to be explored and novel filter criteria that enable the earlier rejection of branches in the search tree. We conduct an extensive experimental evaluation that examines the efficiency and effectiveness of these techniques. Given a time limit of three hours per instance, our best configuration significantly outperforms the `1PlanarTester` and is able to solve more than 98% of the non-planar instances from the well-known `North` and `Rome` graphs with up to 50 vertices. On the whole data set, our best configuration solves 88% of all non-planar `North` graphs and 61% of all non-planar `Rome` graphs, which contain instances with up to 110 vertices.

We give basic definitions in Section 2. In Section 3 we describe our underlying backtracking strategy and the corresponding search tree followed by a description of our traversal strategy. Section 4 is then devoted to the presentation of different branching strategies and in Section 5 we describe multiple filter criteria. In Section 6 we present the results of our experimental evaluation. Finally, we discuss in Section 7 how our backtracking strategy can be adjusted for IC-planarity and NIC-planarity, two subclasses of the 1-planarity problem, and conduct additional experiments for these two problems. We conclude with a brief summary in Section 8.

We remark that Pupyrev [42] simultaneously and independently designed and implemented an algorithm for testing 1-planarity based on a reduction to a SAT-instance.

2 Preliminaries

We assume familiarity with basic graph terminology. Throughout this paper, all graphs are *simple*, i.e., they have neither parallel edges nor loops. We use K_n to denote the complete graph on n vertices and $K_{n,m}$ to denote the complete bipartite graph whose bipartition contains n and m vertices, respectively.

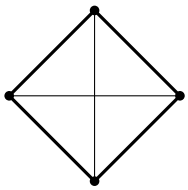

Figure 1: A kite with kite edges in bold.

A *drawing* Γ of a graph $G = (V, E)$ maps every vertex $v \in V$ to a point $\Gamma(v)$ in the plane and every edge $uv \in E$ to a Jordan arc with endpoints $\Gamma(u), \Gamma(v)$ that does not pass through any $\Gamma(w)$ for $w \in V \setminus \{u, v\}$. For simplicity, we identify vertices and edges with their images in a drawing. A *crossing* is a common interior point of two edges. A drawing Γ of a graph G is *1-planar*, if every edge is crossed at most once in Γ. The *planarization* of a drawing Γ is the graph we obtain from Γ by replacing every crossing in Γ with a dummy vertex. A drawing subdivides the plane into connected regions called *faces*, one of which is unbounded and is called *outer face*. Two 1-planar drawings are *equivalent* if they have the same planarization and the same cyclic order of the incident edges around each vertex and each crossing. A 1-*planar embedding* is an equivalence class of 1-planar drawings. A 1-*plane graph* is a graph together with a fixed 1-planar embedding and a *kite* is a 1-plane graph isomorphic to K_4 whose outer face is bounded by a cycle of four vertices and four uncrossed edges, called *kite edges*, while the remaining two edges cross each other; see Figure 1.

It is well known that a 1-planar graph on n vertices has at most $4n - 8$ edges [41]. We call a graph with more than $4n-8$ edges *trivially non-1-planar*. A graph is *biconnected*, if it is connected and remains connected after the removal of any vertex. Recall that a graph is non-planar if and only if it contains a subgraph that is isomorphic to a subdivision of $K_{3,3}$ or K_5 [33]. We call such a subgraph a *Kuratowski-subdivision*.

3 The Backtracking Procedure

In this section we formalize the backtracking strategy we use in the rest of this paper. It is based on a search tree similar to the one of Binucci et al. [7] described in the introduction, whose nodes correspond to partial solutions. Recall that the algorithm of Binucci et al. constructs the search tree based on a global order σ of the crossing candidates. Thus, in their setting, a partial solution corresponds to a bitstring that describes the choices made on a prefix of the order σ. As mentioned in the introduction, the performance seems to crucially depend on σ. We conjecture that, rather than fixing a global order σ on the crossing candidates, it may be beneficial to choose the next crossing candidate to branch on according to criteria that take into account the current partial solution and thus allow different subtrees to explore the same crossing candidate at different depths of the search tree. This may enable the search to focus on critical parts of a partial solution and has the potential to reject infeasible partial solutions more quickly. For this reason, we choose a different representation of partial solutions that encodes the current state more explicitly.

As a graph is 1-planar if and only if each of its biconnected components is 1-planar, we assume in the following that our input graph $G = (V, E)$ is biconnected. Before we describe our backtracking strategy, we introduce some useful notation. For a crossing candidate $\{e, f\}$ with $e = uv$, $f = wy$ let $G_{\{e,f\}}$ denote the graph we obtain by inserting a dummy vertex x subdividing e and f, i.e.,

$G_{\{e,f\}} = (V \cup \{x\}, (E \setminus \{e,f\}) \cup \{ux, vx, wx, yx\})$. Note that such a dummy vertex can either correspond to a crossing or to a touching point, depending on the order of its incident edges. However, the latter does not affect the algorithm as two edges that touch but do not cross can always be redrawn so that they no longer touch without introducing new crossings. We say that the edges ux, vx and wx, yx *stem from* e and f, respectively. For a set C of crossing candidates such that each edge in E is contained in at most one pair of C, we denote by G_C the graph we obtain from G by inserting a dummy vertex as described above for every pair in C.

Let $\bar{E} \subseteq \binom{E}{2}$ be the set containing all crossing candidates $\{e,f\}$ consisting of non-adjacent edges e and f. We encode a partial solution for G as a pair (C,P) with $C, P \subseteq \bar{E}$. The intended meaning is that we have chosen the crossing candidates in C and we may still choose among the crossing candidates in P. Since each edge may receive at most one crossing, we require that for each edge $e \in E$ that appears in a pair $\{e,f\} \in C$, the pair $\{e,f\}$ is the only pair in $P \cup C$ that contains it. We call an edge of G_C *free* if it is contained in a pair in P and we call it *saturated* otherwise. Observe that G is 1-planar if and only if there exists a partial solution (C,P) where G_C is planar. A partial solution (C,P) is *extendable* if there exists a subset $C' \subseteq P$ such that $G_{C \cup C'}$ is planar.

The backtracking procedure incrementally constructs a search tree T starting from the partial solution $(\emptyset, \bar{E})$. We process a node of T representing a partial solution (C,P) as follows. First, if G_C is planar, we return the corresponding 1-planar embedding of G. Second, if G_C is not planar but $P = \emptyset$, we backtrack, since (C,P) cannot be extended to a 1-planar drawing. If neither of these applies, we use a *branching strategy* to compute a finite set $\mathcal{C}$ of child partial solutions (C',P') with $C \subseteq C' \subseteq (C \cup P)$ and $P' \subsetneq P$. The chosen set $\mathcal{C}$ has to be *exhaustive* in the sense that the partial solution (C,P) is extendable if and only if the same holds for at least one child partial solution in $\mathcal{C}$.

We note that this basic framework leaves several degrees of freedom. First, the choice of the branching strategy can have a significant impact on the size of the search tree. Second, in addition to that, the order in which the nodes of T are processed can have a major impact on the time it takes to find a solution. We discuss different options in Section 4.

Third, a major issue stems from the fact that for no-instances the entire tree T needs to be traversed. If we identify that a certain partial solution (C,P) is not extendable, its subtree contains no solution and we can immediately backtrack. Hence, a necessary criterion for extendibility of (C,P) can be used to prune the search tree whenever the criterion is not satisfied. We call such a criterion a *filter*. An example of such a filter is excluding nodes (C,P) where G_C is trivially non-1-planar. We propose additional filters in Section 5.

Another optimization already described by Binucci et al. [7] exploits the fact that kite edges can always be redrawn such that they do not cross any other edge. In particular, when adding a new crossing $\{e,f\}$ to the currently considered partial solution, they insert all edges connecting an endpoint of e to an endpoint of f that are not already present in G and make all these kite edges *uncrossable*, which in our setting corresponds to removing from P all pairs that contain such a kite edge. We note that this can be slightly strengthened by making all edges uncrossable that are contained in a path from an endpoint of e to an endpoint of f whose internal vertices are all of degree 2, as such paths can also always be redrawn arbitrarily closely along e and f without any crossings; see Figure 2. Further note that the insertion of the kite edges enforces that e and f cross rather than touch. The described optimizations help in keeping the search tree small as the number of crossings that have to be considered decreases, and violations of necessary conditions on the extendability to a 1-planar embedding are found earlier during the execution. We thus include them in each of our algorithms.

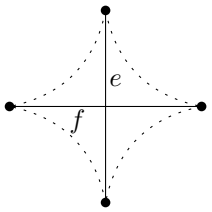

Figure 2: The dotted degree-2-paths can always be redrawn without any crossing.

4 Branching and Traversal Strategies

In this section, we propose different branching strategies for choosing the children of a partial solution (C, P) and discuss different options for the traversal of the resulting search tree. We discuss branching strategies in Section 4.1 and traversal strategies in Section 4.2.

4.1 Branching Strategies

Let (C, P) be a partial solution and assume that G_C is non-planar and $P \neq \emptyset$.

Binary Branching. A basic branching strategy consists of choosing a crossing candidate $x \in P$ and creating children $(C, P \setminus \{x\})$ and $(C \cup \{x\}, P')$, where P' is obtained from P by removing all crossing candidates that contain an edge from x. We refer to such a strategy as *binary branching* and we consider three variants of it. The first is the one also employed by `1PlanarTester` of Binucci et al. [7], where we initially choose a (random) global order σ of the crossing candidates and x is always chosen as the smallest element of P with respect to this order. We refer to this strategy as `Sequential Branching`. For `Random Branching`, we choose x uniformly at random from P and for `KuratowskiFrequency-`k `Branching`, we extract k Kuratowski subdivisions from the non-planar graph G_C and choose x such that the corresponding pair of edges is together contained in a maximum number of these Kuratowski subdivisions. We remark that this can be done in linear time using an algorithm by Chimani, Mutzel and Schmidt [17].

Exhaustive Set Branching. The other major branching strategy we explore is *exhaustive set branching*, where we use an ordered set $P' \subseteq P$ of edge pairs with the property that every 1-planar embedding of G containing the crossings in C also contains at least one crossing from P'. Given such a set $P' = \{p_1, \ldots, p_k\}$, we produce k children (C_i, P_i), where $C_i = C \cup \{p_i\}$ and P_i is obtained from $P \setminus \{p_1, \ldots, p_i\}$ by removing all crossing candidates that contain an edge of p_i. That is, in the ith child, the crossing candidate p_i is chosen and the crossing candidates $p_1, \ldots, p_{i-1}$ are rejected.

Our choices of P' and its ordering described below are guided by which edge pairs are more likely to correspond to "helpful" crossings and thus should be tried with higher priority. In all cases, we use Kuratowski subdivisions of G_C to guide us towards the set P' as, for any Kuratowski subdivision H of G_C, at least one pair of non-adjacent edges that do not belong to the same subdivision path needs to be crossed. We call this set $P(H)$.

In its simplest form, called `KuratowksiSingle Branching`, we extract from G_C a single Kuratowski subdivision H and branch on the set $P(H) \cap P$ with an arbitrary order. Kuratowski subdivisions that are small or have few free edges produce fewer children which can significantly

narrow the search space. Strategy `KuratowskiMulti-`k `Branching` exploits this by extracting a set $\mathcal{H}$ of k Kuratowski subdivisions and choosing the subdivision $H \in \mathcal{H}$ that minimizes $|P(H) \cap P|$ to branch on. To increase the likelihood of trying helpful crossings early, we further order the crossing candidates in $P(H) \cap P$ by the number of Kuratowski subdivisions in $\mathcal{H}$ that contain them.

4.2 Traversal Strategies

The choice of a branching strategy determines the search tree T. However, there are still different options for traversing T. The algorithm `1PlanarTester` of Binucci et al. [7] employs a depth-first search (DFS) to traverse the search tree. This has the effect that a few poor initial choices may necessitate the exploration of a large subtree that does not contain a solution. Especially for larger instances, the algorithm may not recover from this in a timely manner as it has to explore the entire subtree.

An obvious alternative to this is the use of a breadth-first search (BFS), which prioritizes partial solutions with few crossings and may thus even find a 1-planar embedding with the minimum number of crossings. However, due to the large size of T, the corresponding memory requirement is prohibitive as essentially the entire tree needs to be kept in memory.

We thus propose a hybrid approach. We maintain a queue Q with up to k distinct *threads* that correspond to independent parallel depth-first searches. In each step we extract the next thread t from the queue and execute one step of the corresponding DFS. When processing a node (C, T), while Q contains fewer than k threads, instead of moving to a child, we rather insert a new thread into Q that starts a depth-first search at this child. When Q reaches the maximum of k threads, the remaining unvisited children are pushed to the stack of the depth-first search corresponding to t.

In comparison to traversing T by a single DFS, in case of a yes-instance, this may prevent us from fully exploring a very deep unhelpful branch and may allow to find potential solutions earlier during the execution. We note that, for no-instances the whole tree needs to be explored and the choice of the traversal strategy thus does not affect the running time.

5 Filter Criteria

In this section, we describe several filters to determine that a partial solution (C, P) is not extendable and, therefore, allows us to ignore its entire subtree. This has significant potential to speed up the solution speed for both 1-planar and non-1-planar instances.

We first describe the filters that were already used by Binucci et al. [7] and that we employ in all our strategies. We phrase each of them as a necessary condition, whose violation allows to cut the corresponding subtree. The first, called `Planar Structure`, requires that the subgraph of G_C induced by the saturated edges is planar. To strengthen this, we also include the kite edges and the extended kite edges as described at the end of Section 3. The filter `Edge Density` rejects instances where G_C is trivially non-1-planar after adding potentially missing kite edges. We propose three further types of filters.

Partial Planarity. Let (C, P) be a partial solution and let H be the subgraph of G_C that consists of the saturated edges. Partial planarity filters are based on the fact that if (C, P) is extendable, then G_C admits a drawing where H is crossing-free (but all other edges may cross arbitrarily). The corresponding decision problem is known as Partial Planarity. Da Lozzo and Rutter [34] gave a linear-time algorithm, which is however very involved. Therefore, we

instead implement the slower but simpler Hanani-Tutte style algorithm of Schaefer [49]. Due to the high running time of $O(n^6)$, we initially run cheaper tests of weaker necessary properties as described below. For each variant, we assume that all previous variants are used as well.

- `PP1`: A necessary condition for partial planarity is that for each free edge e, the graph $H + e$ is planar. We thus test this for each edge and reject if this is not the case.
- `PP2`: Let $\{e, f\}$ be a pair of free edges in G. Assuming `PP1` has not rejected, we know that $H + e$ and $H + f$ are planar. If $H + \{e, f\}$ is non-planar, we know that e must cross f. In this case, if $\{e, f\} \in P$, we can include this crossing in C and otherwise we can reject the partial solution. We check for each pair of free edges in G whether it is forced in this way.
- `PP3`: Let B be a connected component of $G_C - H$. We call such a subgraph a *bridge* of G_C. Now in a potential 1-planar embedding of G with all crossings from C, each bridge must lie entirely within one face of H. In other words, for the extendability of the partial solution, it is necessary that the graph consisting of H and the bridge B, where B is contracted into a single vertex, is planar.
- `PP4`: Finally, if none of the cheaper tests rejects the partial solution, we run the $O(n^6)$ partial planarity algorithm by Schaefer [49].

  Due to the high running time of `PP4`, we also propose a variant `PP4C`X for some integer $X \geq 1$, where we only run the test with probability $1/X$. In that case, if we detect that G_C is not partially planar, we also run the test on ancestors in the search tree that have not been tested previously in order to find the highest ancestor x whose corresponding planarization is also not partially planar. We can then prune the entire subtree rooted at x. In this way, a single invocation of the test can prune multiple nodes from the search tree.

Separating Cycles (`SC`). Let (C, P) be a partial solution in the search tree T and let $e = uv, f = xy$ be two edges in G that cross each other in the corresponding partial solution; i.e, $\{e, f\} \in C$. Let p be a uv-path in $G - \{e, f\}$ that contains the minimum number of free edges. Let further F be a maximum set of pairwise edge-disjoint paths between x and y such that no path in F shares a vertex with p; see Figure 3 for an illustration. Since e and f cross, the endpoints x and y of f lie on different sides of the cycle K formed by p and e. Thus each path in F needs to cross K. It is thus necessary that each path of F contains an edge that still admits a crossing with an edge of K and that K contains at least $|F|$ free edges.

In our implementation, if the test succeeds for a path p, we temporarily remove it from the graph and repeat the test until there is no longer a path that connects x to y. We execute this test for all pairs $\{e, f\} \in C$.

1-Planarity Obstructions. If G contains a subgraph that is not 1-planar, then G itself is also not 1-planar. For a node (C, P) in the search tree, it is even necessary that G_C contains no non-1-planar subgraph after contracting an arbitrary subset of the saturated edges in G_C. Thus obstructions, such as K_7 or $K_{3,7}$ can emerge in the search tree even if they were not present in G itself.

Since 1-Planarity is NP-complete, it has no finite obstruction set unless P = NP. We thus picked a set of small graphs that are known to be not 1-planar and used an Integer Linear Program (ILP) to determine whether G_C contains one of these graphs as a subgraph after contracting a subset of saturated edges. Initial experiments showed that (1) the ILP is impractically slow and (2) our

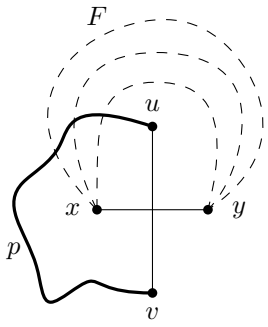

Figure 3: If $e = uv$ crosses $f = xy$, then in any 1-planar drawing, all pairwise edge-disjoint paths between x, y in F (dashed) cross the path p between u and v (bold).

chosen obstructions almost never emerge during the backtracking. While we could test rather efficiently in the beginning whether G itself contains one of the obstructions and thus reject some of the test instances accordingly, we believe that this does not provide any added value to our evaluation. We thus exclude this filter from our benchmarks.

6 Experimental Evaluation

In this section, we experimentally evaluate our backtracking procedure. A concrete instance of our backtracking procedure, called a *configuration* is obtained by choosing a branching strategy and a traversal strategy from Section 4 along with a subset of the filters from Section 5. As some of the strategies have additional parameters, this yields a relatively large number of possible configurations. To address this, we first perform preliminary experiments on a smaller test set and with a short time limit to assess the impact of the individual strategies and filter criteria and to ultimately determine the configuration with the best performance. Afterwards, we compare the performance of this winning configuration with that of the algorithm `1PlanarTester` of Binucci et al. [7] by running it on an extensive test set in a way that closely mimics their experimental setup. As an additional reference point, we also implemented a simple ILP-based algorithm that builds on the ILP formulation of Buchheim et al. [11] for the Exact Crossing Number Problem.

In the following, we first describe the ILP formulation followed by a description of our test data and the experimental setup. Finally we report the results of our evaluation.

6.1 ILP Formulation

In this section we describe our ILP-based algorithm which builds on the ILP formulation of Buchheim et al. [11] for the Exact Crossing Number Problem. Given the input graph G, for each candidate crossing $\{e, f\}$ of edges of G, we introduce a binary variable that indicates whether e and f cross in a solution and add constraints that ensure that every edge is contained in at most one crossing. We then repeat the following steps.

1. If the model is infeasible, reject G as not 1-planar.
2. Otherwise, consider an arbitrary solution of the model and let G' denote the graph obtained by replacing each edge pair that is crossed in the solution with a crossing vertex.

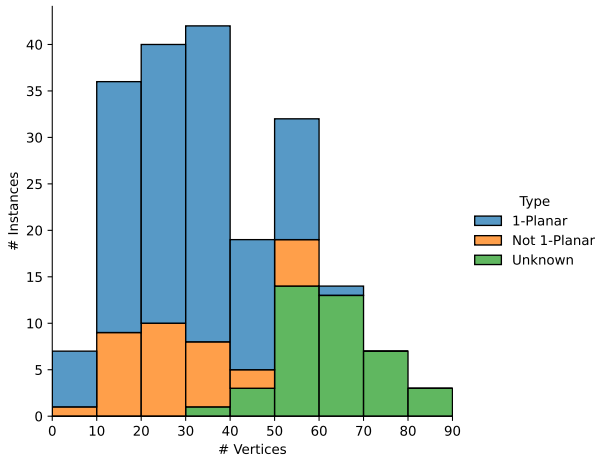

Figure 4: The distribution of instance sizes for the data set `Bicomps-NR-Small`.

(a) If G' is planar, then report G as 1-planar.

(b) Otherwise, consider a Kuratowski-subdivision K of G'. Add a constraint to the model that prohibits K, i.e., ensure that (1) an edge pair is crossed such that K is removed or (2) a crossed edge pair whose crossing vertex is contained in K is "uncrossed". Then continue with Step 1.

Observe that the initial model is always feasible as it does not contain constraints that enforce a crossing and hence setting all variables to 0 is valid. Then we have $G' = G$, which is generally not planar, and thus leads to the addition of constraints in Step 2(b). This is repeated until either the model becomes infeasible (in which case the graph is not 1-planar) or G' is planar and thus a planarization of a 1-planar drawing of G.

We note that this ILP is only meant to serve as a reference point and we consider it beyond the scope of this work to investigate the possibility of combining our filters and strategies with the ILP. A recent work of Chimani and Wagner [18] investigates ILP formulations for 1-planarity based on forbidden crossing patterns.

6.2 Test Data

To make our results comparable to the results by Binucci et al. [7], we also work with the `North` and `Rome` graphs [39], two well-established test sets of graphs that stem from real-world applications. Since planar graphs are trivially 1-planar, we restrict ourselves to the non-planar instances. We let `NR` denote the test set consisting of all non-planar `North` and `Rome` graphs.

Since a graph is 1-planar if and only if each of its biconnected components is 1-planar, we believe that benchmarks including the input size are more representative if one considers each biconnected component individually. Therefore, we further decompose the graphs from `NR` into their non-planar biconnected components for most of our tests. For our initial evaluation of the different configurations of our backtracking procedure, we use a smaller data set `Bicomps-NR-Small` that consists of 100 randomly chosen non-planar biconnected components from the `North` graphs and 100 randomly chosen non-planar biconnected components from the `Rome` graphs. Figure 4 shows

the distribution of the instances in this data set. To eventually evaluate the best configuration, we also use the data sets `Bicomps-N` and `Bicomps-R` containing all non-planar biconnected components from the `North` and `Rome` graphs, respectively.

Finally, as another source of test instances, we consider an additional data set `Random` that consists of four randomly generated graphs for each combination of $n \in \{10, 11, \ldots, 50\}$ and $p \in \{0.01, 0.02, \ldots, 1\}$. Each graph in the data set has n vertices, where each edge is included independently with probability p.

6.3 Experimental Setup

Our implementation is written in C++ and uses the Open Graph Drawing Framework (OGDF) [13] for graph representation and algorithms. The code was compiled using GCC version 12.2.0 and optimization `-O3`. Each experiment was run on a single core of an Intel Xeon E5-2690v2 CPU (3.0 GHz) with a memory limit of 10 GiB and running Linux Kernel version 6.1.0-37. We use Gurobi version 12.0.2 as a solver for our reference ILP.

6.4 Evaluation of Backtracking Strategies

In this section, we run preliminary experiments on the data set `Bicomps-NR-Small` with a time limit of ten minutes to determine the best configuration for our backtracking procedure. Each configuration includes kite edges and the basic filters for edge density and non-planarity of the saturated subgraph, as these optimizations are cheap in terms of running time. Note that our main criterion for comparing the performance of algorithms is the number of solved instances within a fixed timelimit.

Branching Strategies. We first evaluate the different strategies proposed in Section 4.1 for choosing the crossing candidates to branch on in every node of the search tree with a DFS as traversal strategy and only the basic filters; see Figure 5 for the results. Unsurprisingly, the two binary branching strategies, which do not consider the structure of the graph in any way (`Sequential` and `Random`) perform the worst. In comparison, the binary branching strategy `KuratowskiFrequency`, which prioritizes edge pairs that appear in many Kuratowski subdivisions, solves roughly 50% more instances than `Sequential`. Unlike `Sequential` and `Random`, `KuratowskiFrequency` only considers edge pairs that belong to a Kuratowski subdivision, which also improves its performance for no-instances; see Figure 5. Considering more than 100 Kuratowski subdivisions does not make a notable difference for this strategy.

The two exhaustive set branching strategies (`KuratowskiSingle` and `KuratowskiMulti`) clearly outperform the binary branching strategies. This is due to the fact that the search space significantly shrinks, as the branch where no edge pair of the chosen Kuratowski subdivision is crossed can be omitted. Interestingly, the ILP formulation also performs significantly better than the binary branching strategies and is on par with `KuratowskiSingle`. It performs better than `KuratowskiSingle` on yes-instances, but is only able to recognize no-instances that are trivially non-1-planar. The clear overall winner is the strategy `KuratowskiMulti`, where we consider multiple Kuratowski subdivisions and branch over the one with the lowest number of free edge pairs, prioritizing edge pairs that occur in many Kuratowski subdivisions. While `KuratowskiMulti` clearly benefits from extracting 1000 Kuratowski subdivions instead of 100, increasing this number to 10000 makes no notable difference. This could be at least partly due to the fact that the graphs that have such a high number of Kuratowski subdivisions are too large to be solved within the

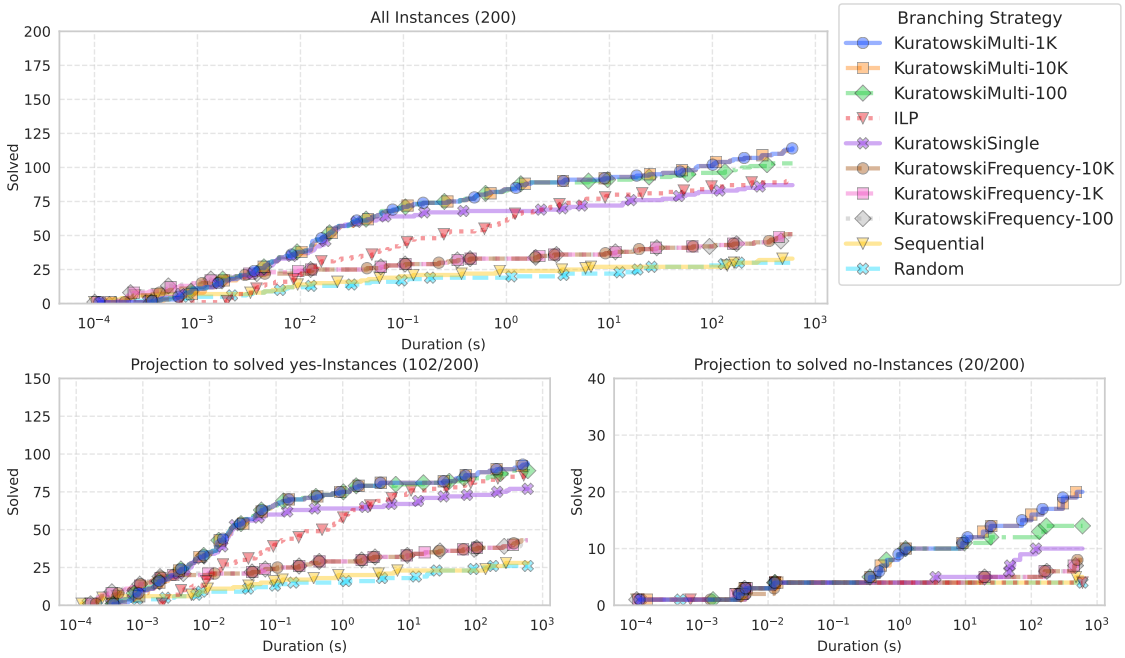

Figure 5: The number of solved instances over time on the 200 graphs of `Bicomps-NR-Small` for each branching strategy from Section 4.1. The suffix *-x* for `KuratowskiMulti` and `KuratowskiFrequency` denotes the number of extracted Kuratowski subdivisions.

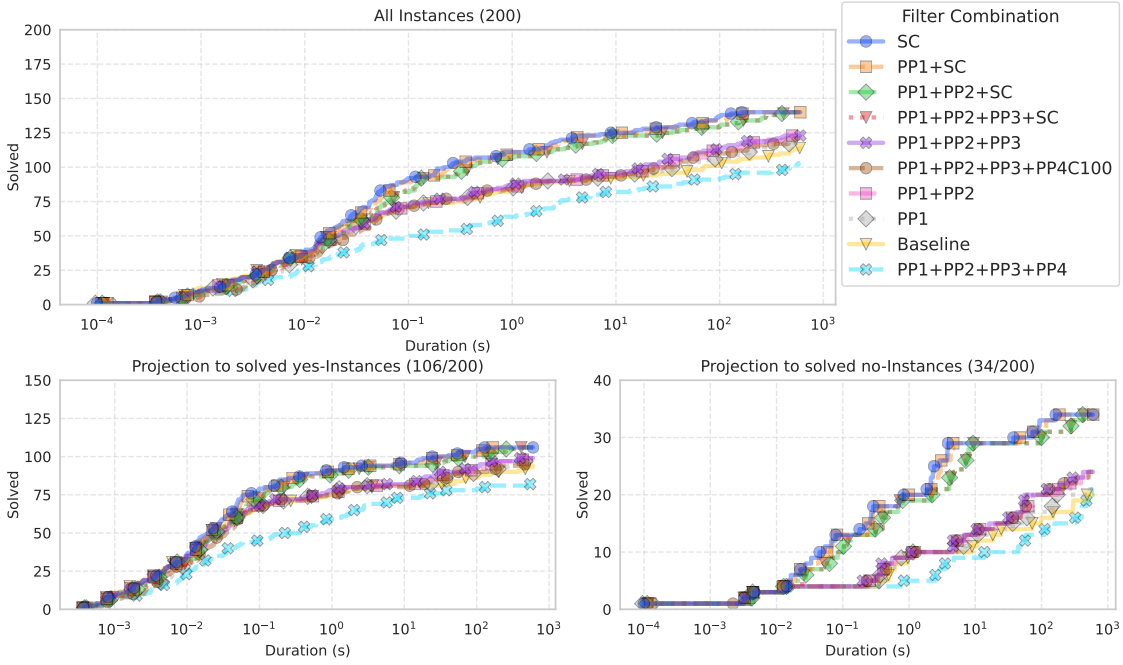

Figure 6: The number of solved instances over time on the 200 graphs of `Bicomps-NR-Small` for different combinations of filters from Section 5.

ten minute timelimit. However, since extracting more Kuratowski subdivisions comes at a higher cost of running time, we choose `KuratowskiMulti-1K` as the winning branching strategy for the subsequent experiments.

Filter Criteria. Next, we consider the filter criteria introduced in Section 5. As a baseline, we use the winning branching strategy `KuratowskiMulti-1K` from above. Figure 6 illustrates the solved instances over time for different (combinations of) filters. Moreover, Table 1 shows the number of solved instances and the corresponding *success rate* of each filter, i.e., the number of nodes in the search tree that were rejected by the filter divided by the total number of its invocations. The simplest partial planarity filter (`PP1`) alone rejects almost 60% of all nodes in the search tree it is invoked on and this already increases the number of solved instances by roughly 5% compared to the baseline. Additionally enabling `PP2` and `PP3` only slightly improves the number of solved instances. Unsurprisingly, due to the $O(n^6)$ running time of `PP4`, also enabling this filter is detrimental for the algorithm due to its high execution time. Running the filter only 1% of the time (`PP1+PP2+PP3+PP4C100` in Table 1) raises the overall success rate of the filter over 100%, since we retrospectively test ancestors of nodes where the filter was successful and prune the search tree accordingly (see Section 5). However, it is overall still better to not enable this filter. Instead, using the filter `SC` yields a significant improvement and allows the backtracking procedure to solve roughly 23% more instances than the baseline overall, independent from what other filters are enabled.

Filter Combination	Solved	Filter Success Rate				
		PP1	PP2	PP3	PP4	SC
Baseline	114	-	-	-	-	-
PP1	120	0.58	-	-	-	-
PP1+PP2	123	0.37	0.38	-	-	-
PP1+PP2+PP3	123	0.27	0.35	0.51	-	-
PP1+PP2+PP3+PP4	103	0.2	0.27	0.41	0.26	-
PP1+PP2+PP3+PP4C100	121	0.26	0.33	0.48	1.29	-
SC	140	-	-	-	-	0.9
PP1+SC	140	0.08	-	-	-	0.89
PP1+PP2+SC	140	0.08	0.14	-	-	0.87
PP1+PP2+PP3+SC	140	0.08	0.14	0.2	-	0.84

Table 1: The number of solved instances and filter success rates for different combinations of the filters in Section 5 on the data set `Bicomps-NR-Small`. The left-to-right order of the columns indicates the order in which the filters are invoked in the implementation. Note that the ratio greater than 1 for `PP4C100` stems from the pruning of ancestors; see Section 5.

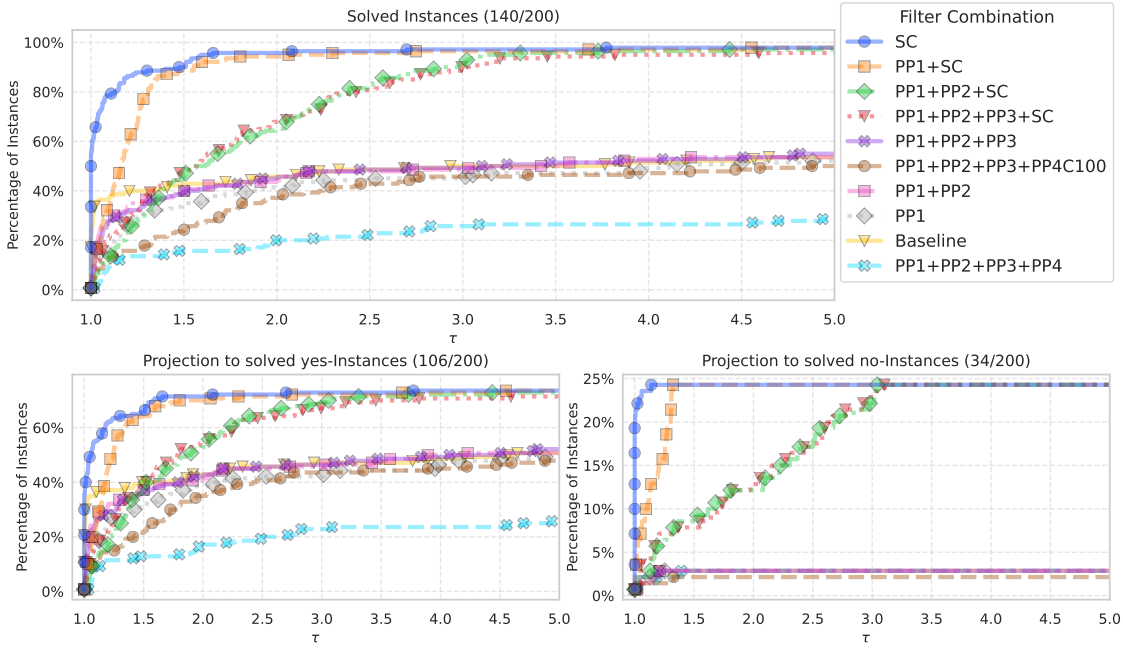

Figure 7: The performance profiles [22] for different filter combinations with the computing time as the performance metric. For each filter combination A, the plot shows the percentage of instances where the computing time of A is at most by a factor of τ slower than the fastest time among all combinations.

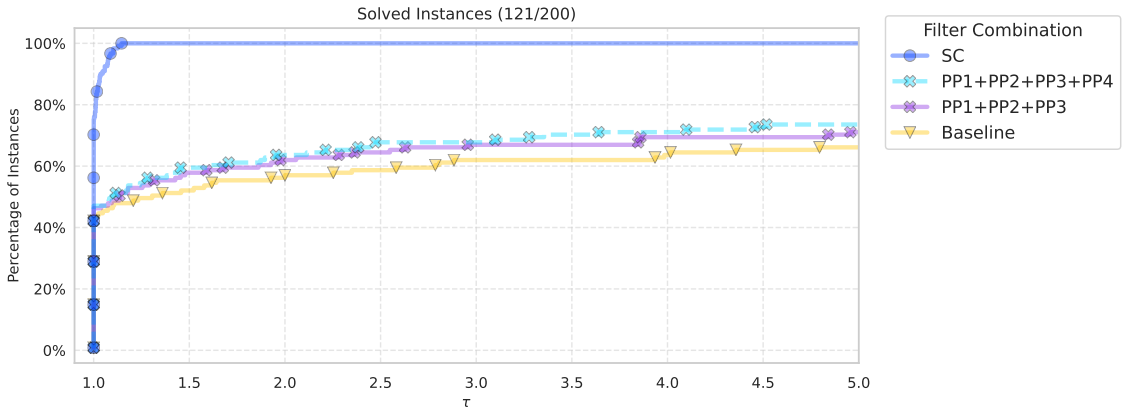

Figure 8: The performance profiles [22] for different filter combinations with the number of processed nodes (instead of the running time) as the performance metric. For each filter combination A, the plot shows the percentage of instances where the number of processed nodes of A is at most by a factor of τ larger than the lowest number of processed nodes among all combinations. Instead of a time limit, the algorithms were given a limit of 5000 nodes in the search tree that can be processed for each instance.

In order to evaluate the filter combinations in more detail, we additionally use *performance profiles* [22]. For a set of algorithms and a specific performance metric (such as computing time), a corresponding performance profile shows for each algorithm A for what percentage of instances the performance of A lies within a certain distance of the best performance among all algorithms. In Figure 7, we first consider the computing time as the performance metric. Clearly, the combinations including the filter `SC` significantly outperform all combinations where `SC` is disabled. In fact, the algorithm has the best performance if only the filter SC is enabled: it is the fastest combination for over 50% of the solved instances and its running time differs at most by a factor of 1.5 from the fastest combination for around 90% of the instances that are solved by any of the algorithms. This indicates that `SC` dominates the other filters, i.e., it most likely rejects many partial solutions in the search tree earlier than the other filters would. This is further supported by the fact that the success rate of the other filters shrinks significantly if combined with `SC`, despite the fact that `SC` is invoked last; see Table 1. Overall, `SC` alone rejects around 90% of all search tree nodes it is invoked on.

Altogether, this suggests that enabling only the filter `SC` is the winning configuration. However, it should be noted that this evaluation is potentially unfair towards the more costly filters such as `PP4`. Our implementation has a rather high time complexity of $O(n^6)$, which likely prohibits the algorithm from exploring a significant number of nodes of the search tree within the time limit. Since, theoretically, the filter `PP4` could be implemented with linear running time [34], it would be premature to reject `PP4`.

To better assess the potential of the filters, independent of their associated costs in terms of the running time of our implementation, we additionally consider performance profiles in Figure 8, where we consider as the performance metric the number of processed nodes instead of the running time. Instead of a time limit, the implementations are given a limit of 5000 nodes in the search

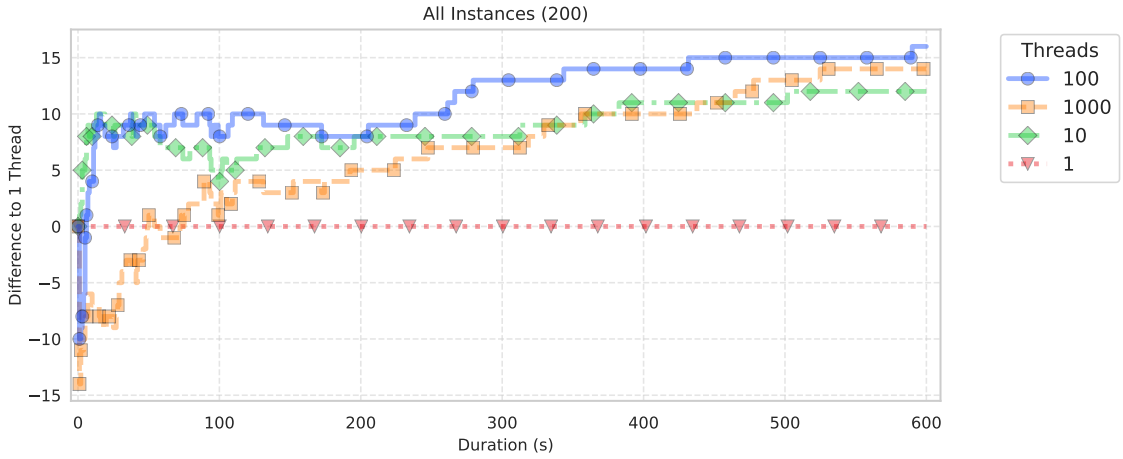

Figure 9: The performance of different thread counts for branching strategy `KuratowskiMulti-1K` with filter `SC` on the data set `Bicomps-NR-Small`. The plot shows the absolute difference in the number of solved instances over time to the baseline using only a single thread.

tree that can be processed. It is not surprising that, if running time is disregarded, it is always better to use as many filters as possible. However, Figure 8 clearly shows that, even in this case, `PP4` only offers a marginal improvement over the simpler variants `PP1`-`PP3` and `SC` alone performs significantly better than all other filters. This indicates that even a linear-time implementation of `PP4` would most likely not yield a meaningful improvement of the backtracking procedure. Moreover, Figure 8 shows that the configuration using only the filter `SC` processes at most 15% more nodes in the search tree than the other combinations. We thus exclusively use `SC` in our winning configuration.

Traversal Strategies. Finally, we investigate the effect of threads (see Section 4.2) on the backtracking algorithm. Using a single thread (which corresponds to a DFS in the search tree) leads to 140 of the 200 instances solved. As shown in Figure 9, using 10 or 100 threads improves the performance of the algorithm, for an additional 16 instances solved for 100 threads. However, increasing the number of threads to 1000 worsens the performance. This is because using more threads also increases the memory usage, which causes the algorithm to exceed the 10 GiB memory limit and to consequently abort on some instances. As expected, the number of threads makes no difference for no-instances, as threads do not shrink the search space. However, using more than one thread is often helpful for identifying yes-instances early.

6.5 Evaluation of the Best Configuration

Our experiments in the previous section indicate that the configuration using the branching strategy `KuratowskiMulti-1K` with filter `SC` and 100 threads performs the best. We implemented a less flexible but more optimized version of our backtracking procedure that is tailored specifically to

Group	#	Group Median			Solved		1-Planar %	
		Density	Proc. Nodes	Time (s)	#	%	Yes	No
Bicomps-R 1-10	9	1.57	2	0.0	9	100.0	100.0	0.0
Bicomps-R 11-20	250	1.47	3	0.0	250	100.0	100.0	0.0
Bicomps-R 21-30	1296	1.39	16	0.01	1296	100.0	99.8	0.2
Bicomps-R 31-40	1843	1.43	519	0.61	1754	95.2	95.0	0.2
Bicomps-R 41-50	1358	1.44	17542	22.16	935	68.9	68.9	0.0
Bicomps-R 51-60	1227	1.44	56964	88.31	396	32.3	32.3	0.0
Bicomps-R 61-70	1126	1.45	55804	120.29	92	8.2	8.2	0.0
Bicomps-R 71-80	929	1.47	31379	80.33	17	1.8	1.8	0.0
Bicomps-R 81-90	214	1.45	29988	88.57	3	1.4	1.4	0.0
Bicomps-R 91-100	1	1.42	–	–	0	0.0	0.0	0.0
Bicomps-R	8253	1.44	386	0.41	4752	57.6	57.5	0.1
Bicomps-N 1-10	56	2.05	4	0.0	56	100.0	92.9	7.1
Bicomps-N 11-20	124	2.0	22	0.01	124	100.0	60.5	39.5
Bicomps-N 21-30	94	1.9	43	0.02	94	100.0	48.9	51.1
Bicomps-N 31-40	52	1.81	159	0.16	42	80.8	50.0	30.8
Bicomps-N 41-50	30	1.48	67	0.11	28	93.3	73.3	20.0
Bicomps-N 51-60	44	1.92	208	0.51	17	38.6	11.4	27.3
Bicomps-N 61-70	21	1.92	767	1.88	2	9.5	4.8	4.8
Bicomps-N 71-80	2	1.58	40	0.09	2	100.0	100.0	0.0
Bicomps-N 81-90	4	1.64	1435	4.77	2	50.0	50.0	0.0
Bicomps-N 91-100	2	1.52	94376	344.72	2	100.0	100.0	0.0
Bicomps-N	429	1.9	27	0.01	369	86.0	54.3	31.7

Table 2: The results for the data sets `Bicomps-N` and `Bicomps-R`, consisting of the non-planar biconnected components of the `North` and `Rome` graphs, respectively, with a time limit of 15 minutes. Note that the medians for the number of processed nodes and the running time only take solved instances into account. The last two columns show the proportion of the total instances that were labeled as 1-planar and not 1-planar, respectively.

this setting.[1] In particular, this implementation requires less memory and can afford to use 1000 threads without running into the memory limit.

Running this algorithm on all non-planar biconnected components of the `North` and `Rome` graphs with a time limit of 15 minutes shows a clear dependency between the running time and the size of the graph. While almost all instances up to 30 vertices are solved, this rate drops to around 70% for graphs of size 50. Overall, our algorithm solves 58% of all non-planar biconnected components of `Rome` graphs and 86% of those from the `North` graphs. Table 2 gives a detailed overview.

To compare our algorithm with the algorithm `1PlanarTester` of Binucci et al. [7], we also followed their experimental setup and ran our algorithm on all non-planar `North` and `Rome` graphs using the same time limit of three hours. Table 3 shows the details of the outcome. We note that, compared to the previous experiment, the correlation between running time and instance size is less pronounced as some instances may decompose into multiple smaller biconnected components that can be processed independently. Our implementation is able to solve all instances with up to 30 vertices within the time limit, with the median computing time being less than 0.1 seconds.

[1] This implementation will be part of the OGDF (`https://www.ogdf.uni-osnabrueck.de/`) in the release following Foxglove (2025.10).

Group	#	Group Median			Solved			1-Planar %	
		Density	Proc. Nodes	Time (s)	#	%	% [7]	Yes	No
Rome 10-20	91	1.5	6	0.0	91	100.0	91.2	100.0	0.0
Rome 21-30	164	1.36	12	0.0	164	100.0	69.5	100.0	0.0
Rome 31-40	1346	1.31	34	0.02	1342	99.7	(43.8)	99.2	0.5
Rome 41-50	1349	1.33	389	0.4	1295	96.0	(37.8)	95.0	1.0
Rome 51-60	1054	1.33	4602	6.74	867	82.3	–	82.2	0.1
Rome 61-70	1102	1.34	25822	31.54	682	61.9	–	61.8	0.1
Rome 71-80	1023	1.33	89396	134.91	410	40.1	–	40.1	0.0
Rome 81-90	744	1.35	103165	170.02	133	17.9	–	17.9	0.0
Rome 91-100	1376	1.35	59564	116.79	110	8.0	–	8.0	0.0
Rome 101-110	4	1.28	–	–	0	0.0	–	0.0	0.0
Rome	8253	1.33	618	0.73	5094	61.7	–	61.4	0.3
North 10-20	121	1.8	9	0.0	121	100.0	73.6	72.7	27.3
North 21-30	69	1.67	41	0.02	69	100.0	39.1	60.9	39.1
North 31-40	55	1.65	66	0.02	52	94.5	38.2	49.1	45.5
North 41-50	52	1.45	276	0.26	51	98.1	(18.8)	65.4	32.7
North 51-60	48	1.89	2579	2.24	26	54.2	–	20.8	33.3
North 61-70	33	1.9	594	0.32	17	51.5	–	18.2	33.3
North 71-80	14	1.3	38	0.0	11	78.6	–	71.4	7.1
North 81-90	18	1.62	70	0.06	17	94.4	–	44.4	50.0
North 91-100	13	1.56	10515	9.45	10	76.9	–	38.5	38.5
North	423	1.73	47	0.01	374	88.4	–	54.4	34.0

Table 3: Our results on the non-planar instances from the graph benchmark sets `North` and `Rome` with a time limit of three hours. Note that the medians for the number of processed nodes and the running time only take solved instances into account. For each group, the multicolumn "Solved" shows the number of instances we solved ("#"), the corresponding percentage of solved instances ("%"), and the percentage of instances the `1PlanarTester` by Binucci et al. [7] solved in the corresponding group within the same time limit ("% [7]"). For the latter, percentages in parantheses indicate that Binucci et al. [7] only evaluated a subset of the instances in the corresponding group. The last two columns show the proportion of the total instances that were labeled as 1-planar and not 1-planar, respectively.

While our algorithm still solves more than 95% of all instances with up to 50 vertices, this ratio notably decreases for larger graphs, especially for the `Rome` graphs. Here, we still solve around 62% of the instances with 61 to 70 vertices, but only 8% of the instances with 91 to 100 vertices. Overall, our implementation solves around 62% of the 8523 non-planar `Rome` graphs. In contrast, we solve almost 90% of all non-planar `North` graphs – our algorithm even manages most of the instances with 71 to 100 vertices. The data in Table 3 shows that our algorithm significantly outperforms `1PlanarTester` on this dataset. For example, while `1PlanarTester` solves around 56% of all non-planar `North` graphs with up to 40 vertices, we solve around 99% of them. In fact, each instance of the data set that `1PlanarTester` solves within the three hour timelimit is solved in less than a second by our algorithm.

6.6 Evaluation on Random Graphs

Balloni, Di Battista, and Patrignani [5] recently studied the "tipping point" of planarity in small to medium-sized random graphs, that is the transition between planarity and non-planarity with increasing density. While it is known that asymptotically this transition occurs at density $\frac{1}{2}$ [35, 40], the experiments by Balloni et al. revealed a fairly sharp transition even for smaller graphs.

In this section, we conduct a similar study for 1-planarity. To this end, we run additional experiments on the randomly generated graphs from the data set `Random`. Figure 10 illustrates the results of `1PlanarTester` on this data set with a time limit of one hour per instance. The instances are arranged based on their corresponding values of n and p, with color-coding indicating the algorithm's output. The different types of solved instances exhibit clearly visible strips. Unsurprisingly, the graphs exhibit a sharp transition to trivial no-instances where $p \cdot \binom{n}{2} \approx 4n - 8$. Interestingly, the transition between 1-planar and non-1-planar graphs seems to be even more distinct than the transition between planar and non-planar graphs. Figure 10 furthermore suggests that the instances that are difficult for `1PlanarTester` to solve lie exactly at the boundary between 1-planar graphs and non-1-planar graphs. In comparison, instances that are close to being planar and instances that are close to being trivially non-1-planar are solved much more quickly. We conclude that further research should focus on instances at the tipping point between 1-planar graphs and non-1-planar graphs as these are the most challenging for the current algorithm. For this reason, we believe that focusing on these instances may reveal additional insights that are likely to increase the overall performance of the algorithm.

7 IC- and NIC-planarity

In this section we explain how to modify our algorithm such that it recognises two subclasses of the 1-planar graphs and evaluate the performance. A graph is *IC-planar* (where IC stands for *independent crossings*) if it admits a 1-planar drawing in which no two crossed edges share an endpoint. A graph is *NIC-planar* (where NIC stands for *near independent crossings*) if it admits a 1-planar drawing in which any two distinct pairs of crossing edges share at most one endpoint. Observe that the IC-planar graphs are a proper subclass of the NIC-planar graphs. For both graph classes the corresponding recognition problem is known to be NP-hard [10, 4].

Observe that when testing IC- or NIC-planarity, the decision to cross two edges generally results in more edge pairs not being allowed to cross compared to the 1-planarity variant. Thus to recognize these two graph classes we modify our branching algorithm for testing 1-planarity as follows. Let T be a search tree computed by `1PlanarTester` and let (C, P) be a partial solution in T. For every child (C', P') in T we remove from P' all edge pairs that share two endpoints with a pair in $C' \setminus C$. For testing IC-planarity we further remove all edge pairs from P' that contain an edge that shares an endpoint with an edge contained in a pair in $C' \setminus C$.

We remark that for testing IC-planarity it does not suffice to process biconnected components independently as we have to guarantee that cut-vertices are incident to a crossing in at most one component; see Figure 11. Instead, non-biconnected graphs can be processed as follows. For every input graph G we consider its *block-cut tree* B, which contains one node per biconnected component in G and one node per cut-vertex in G such that a node that corresponds to a biconnected component H and a node that corresponds to a cut-vertex c are adjacent if and only if c is contained in H. We root B at an arbitrary cut-vertex of G. For a cut-vertex c in G we denote the subtree of B rooted at the node corresponding to c by B_c. Further we denote the subgraph of G induced by all cut-vertices and biconnected components corresponding to nodes in B_c by $G[B_c]$.

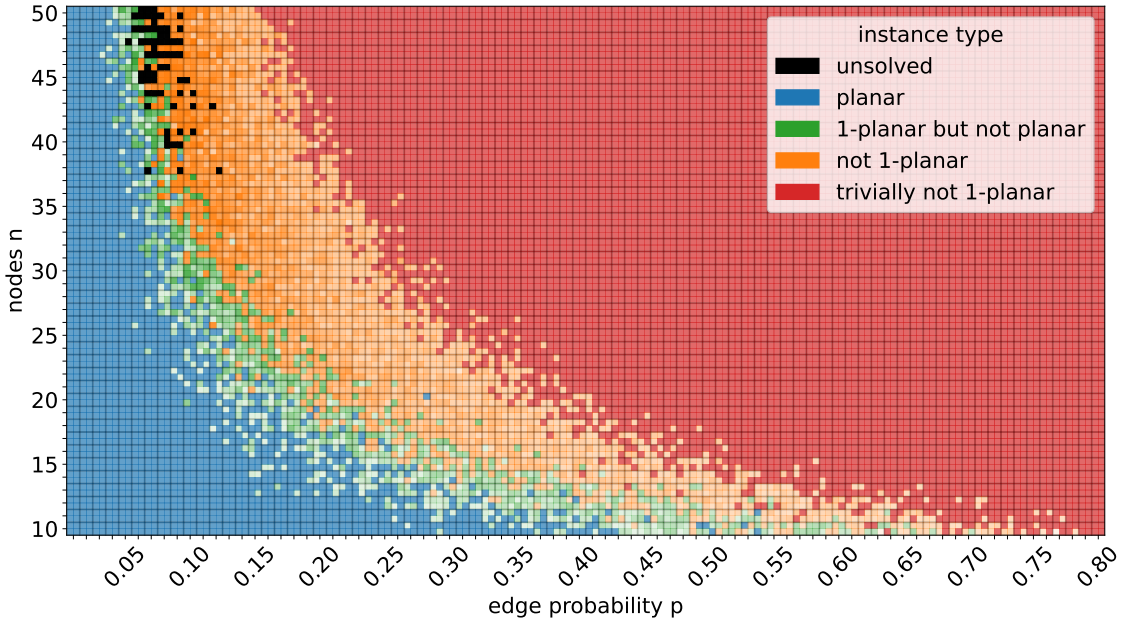

Figure 10: Our results on the data set `Random` with a time limit of one hour per instance. Each 2x2 square corresponds to one combination of n and p and the four cells contained in the square represent the four corresponding instances of the combination. The cells are colored based on the type of the instance, where black cells represent instances that were not solved within the time limit. For all non-trivial instances (i.e., "1-planar but not planar" and "not 1-planar"), the alpha-values of the cells correspond to the running time needed to solve the corresponding instance, at a logarithmic scale. More transparent cells thus represent simpler instances, while more opaque cells took longer to solve.

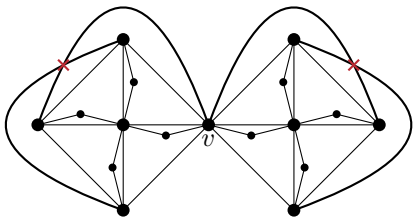

Figure 11: A non-biconnected graph G with cut-vertex v. Each of the two biconnected components of G consists of a K_5 plus a subdivided copy of each of the four edges incident to a designated vertex distinct from v. In any 1-planar embedding of the biconnected components of G an edge incident to v is crossed. Thus G is not IC-planar although both its biconnected components are.

Group	#	Group Median			Solved		IC-Planar %	
		Density	Proc. Nodes	Time (s)	#	%	Yes	No
Bicomps-R 1-10	9	1.57	2	0.0	9	100.0	100.0	0.0
Bicomps-R 11-20	250	1.47	3	0.0	250	100.0	94.0	6.0
Bicomps-R 21-30	1296	1.39	16	0.0	1296	100.0	86.8	13.2
Bicomps-R 31-40	1843	1.43	354	0.12	1843	100.0	62.5	37.5
Bicomps-R 41-50	1358	1.44	3274	1.59	1358	100.0	34.6	65.4
Bicomps-R 51-60	1227	1.44	12541	9.61	1202	98.0	18.1	79.9
Bicomps-R 61-70	1126	1.45	32292	34.93	1020	90.6	6.2	84.4
Bicomps-R 71-80	929	1.47	35460	56.95	734	79.0	2.3	76.7
Bicomps-R 81-90	214	1.45	56555	101.6	140	65.4	2.3	63.1
Bicomps-R 91-100	1	1.42	–	–	0	0.0	0.0	0.0
Bicomps-R	8253	1.44	1719	0.88	7852	95.1	40.1	55.1
Bicomps-N 1-10	56	2.05	4	0.0	56	100.0	58.9	41.1
Bicomps-N 11-20	124	2.0	19	0.0	124	100.0	42.7	57.3
Bicomps-N 21-30	94	1.9	27	0.01	94	100.0	35.1	64.9
Bicomps-N 31-40	52	1.81	36	0.01	52	100.0	36.5	63.5
Bicomps-N 41-50	30	1.48	35	0.03	30	100.0	63.3	36.7
Bicomps-N 51-60	44	1.92	63	0.09	44	100.0	6.8	93.2
Bicomps-N 61-70	21	1.92	191	0.26	21	100.0	4.8	95.2
Bicomps-N 71-80	2	1.58	36	0.05	2	100.0	100.0	0.0
Bicomps-N 81-90	4	1.64	219	0.3	4	100.0	50.0	50.0
Bicomps-N 91-100	2	1.52	141862	252.38	2	100.0	0.0	100.0
Bicomps-N	429	1.9	27	0.01	429	100.0	38.5	61.5

Table 4: Our results for testing the IC-planarity of Bicomps-N and Bicomps-R with a time limit of 15 minutes, analogous to Table 2.

For a biconnected component C let $p(C)$ be the cut-vertex corresponding to the parent of C in B and let $b(C)$ be the set of cut-vertices corresponding to children of C in B. In the following we describe a traversal of B during which we mark a cut-vertex c as *claimed* if it is incident to a crossing in every IC-planar embedding of $G[B_c]$.

Initially all cut-vertices are unclaimed. We traverse B bottom-up and check for every biconnected component C whether it admits an IC-planar drawing in which no claimed cut-vertex in $b(C)$ is incident to a crossing. In the negative case we reject. In the positive case we check whether C admits an IC-planar drawing in which neither the claimed cut-vertices in $b(C)$ nor $p(C)$ is incident to a crossing. In the negative case if $p(C)$ is already claimed we reject; otherwise we mark $p(C)$ as claimed. Observe that if we do not reject, after processing all biconnected components we can construct an IC-planar drawing of G by merging the computed IC-planar drawings of the biconnected components. Moreover, note that we process each biconnected component at most twice.

Analogous to Section 6.5, we evaluate our testers for IC-planarity and for NIC-planarity on the data sets Bicomps-N and Bicomps-R with a time limit of 15 minutes. As shown in Table 4, our IC-planarity tester solves all instances that stem from the North graphs and the vast majority of instances that stem from the Rome graphs and thus performs significantly better than the 1-planarity test on the same data set (cf. Table 2). Interestingly, when testing NIC-planarity, slightly more instances of Bicomps-N but fewer instances of Bicomps-R could be solved within

Group	#	Group Median			Solved		NIC-Planar %	
		Density	Proc. Nodes	Time (s)	#	%	Yes	No
`Bicomps-R` 1-10	9	1.57	2	0.0	9	100.0	100.0	0.0
`Bicomps-R` 11-20	250	1.47	3	0.0	250	100.0	98.8	1.2
`Bicomps-R` 21-30	1296	1.39	16	0.01	1296	100.0	97.8	2.2
`Bicomps-R` 31-40	1843	1.43	555	0.49	1728	93.8	90.0	3.7
`Bicomps-R` 41-50	1358	1.44	11052	14.23	799	58.8	58.4	0.4
`Bicomps-R` 51-60	1227	1.44	41818	58.92	283	23.1	23.1	0.0
`Bicomps-R` 61-70	1126	1.45	31349	64.65	63	5.6	5.6	0.0
`Bicomps-R` 71-80	929	1.47	43421	90.05	14	1.5	1.5	0.0
`Bicomps-R` 81-90	214	1.45	127330	285.83	3	1.4	1.4	0.0
`Bicomps-R` 91-100	1	1.42	–	–	0	0.0	0.0	0.0
`Bicomps-R`	8253	1.44	239	0.22	4445	53.9	52.6	1.3
`Bicomps-N` 1-10	56	2.05	4	0.0	56	100.0	71.4	28.6
`Bicomps-N` 11-20	124	2.0	19	0.0	124	100.0	51.6	48.4
`Bicomps-N` 21-30	94	1.9	40	0.02	94	100.0	46.8	53.2
`Bicomps-N` 31-40	52	1.81	267	0.21	50	96.2	42.3	53.8
`Bicomps-N` 41-50	30	1.48	39	0.08	26	86.7	66.7	20.0
`Bicomps-N` 51-60	44	1.92	74	0.21	22	50.0	11.4	38.6
`Bicomps-N` 61-70	21	1.92	98	0.26	2	9.5	4.8	4.8
`Bicomps-N` 71-80	2	1.58	62	0.11	2	100.0	100.0	0.0
`Bicomps-N` 81-90	4	1.64	1230	2.83	2	50.0	50.0	0.0
`Bicomps-N` 91-100	2	1.52	198097	647.99	2	100.0	100.0	0.0
`Bicomps-N`	429	1.9	27	0.01	380	88.6	47.1	41.5

Table 5: Our results for testing the NIC-planarity of `Bicomps-N` and `Bicomps-R` with a time limit of 15 minutes, analogous to Tables 2 and 4.

the given time limit compared to 1-planarity; see Table 5. A possible reason for this is that, for IC-planarity, the additional constraints shrink the search space significantly, whereas for NIC-planarity, some simple yes-instances turn into difficult no-instances whose search space must be explored exhaustively.

8 Conclusion

We proposed a new backtracking framework for exact 1-planarity testing and investigated the impact of different branching strategies, filter criteria, and traversal strategies on its effectiveness. Our experiments show that some of our optimizations massively decrease the search space for such algorithms and thus greatly enhance their performance. In particular, using Kuratowski subdivisions to guide the search more than doubles the number of solved instances compared to the baseline that uses binary branching and our best filter, `SC`, rejects around 90% of the nodes in the search tree it is invoked on; it thus offers a significant, exponential decrease of the search space. Our best configuration combines this with threads that allow to search multiple parts of the search tree in parallel and altogether yields a practical 1-planarity testing and embedding algorithm that significantly outperforms its competitor `1PlanarTester` [7] and solves most instances from our test set with up to 50 vertices within seconds. Finally we described how to use our backtracking framework for testing IC- and NIC-planarity. The corresponding performance evaluation shows that with our heuristics, IC-planarity can be tested quite efficiently in practice.

An interesting direction for future research is to further investigate structural obstructions to 1-planarity. As described in Section 5 we tested for a set of small non-1-planar graphs whether they emerge as subgraphs during the backtracking but experiments showed that this is rarely the case. In stark contrast, our best filter `SC`, which is also based on structural obstructions, significantly reduced the number of visited nodes in the search tree. It is thus interesting to search for further families of efficiently testable structural obstructions that help to decrease the search space.

Another direction for future work is to integrate our strategies and filters into the ILP. While the ILP struggles with no-instances, it already outperforms many of the simpler backtracking-procedures on yes-instances and it would be interesting to see whether its performance can be further improved by applying our branching and traversal strategies and including our filters as separation routines. Conversely, it is an interesting question whether a relaxation of the ILP can be used to guide the branching during the backtracking procedure. Similarly, it would be interesting to explore whether strategies of our backtracking framework can be applied to the SAT-formulation by Pupyrev [42] or vice versa.

References

[1] Patrizio Angelini, Michael A. Bekos, Michael Kaufmann, and Thomas Schneck. Efficient generation of different topological representations of graphs beyond-planarity. *Journal of Graph Algorithms and Applications*, 24(4):573–601, 2020. doi:10.7155/jgaa.00531.

[2] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Andreas Gleißner, Kathrin Hanauer, Daniel Neuwirth, and Josef Reislhuber. Outer 1-planar graphs. *Algorithmica*, 74(4):1293–1320, 2016. doi:10.1007/s00453-015-0002-1.

[3] Christopher Auer, Franz J. Brandenburg, Andreas Gleißner, and Josef Reislhuber. 1-planarity of graphs with a rotation system. *Journal of Graph Algorithms and Applications*, 19(1):67–86, 2015. doi:10.7155/jgaa.00347.

[4] Christian Bachmaier, Franz J. Brandenburg, Kathrin Hanauer, Daniel Neuwirth, and Josef Reislhuber. NIC-planar graphs. *Discret. Appl. Math.*, 232:23–40, 2017. doi:10.1016/J.DAM.2017.08.015.

[5] Emanuele Balloni, Giuseppe Di Battista, and Maurizio Patrignani. A tipping point for the planarity of small and medium sized graphs. In David Auber and Pavel Valtr, editors, *28th International Symposium on Graph Drawing and Network Visualization (GD '20)*, volume 12590 of *Lecture Notes in Computer Science*, pages 181–188. Springer, 2020. doi:10.1007/978-3-030-68766-3_15.

[6] Michael J. Bannister, Sergio Cabello, and David Eppstein. Parameterized complexity of 1-planarity. *Journal of Graph Algorithms and Applications*, 22(1):23–49, 2018. doi:10.7155/jgaa.00457.

[7] Carla Binucci, Walter Didimo, and Fabrizio Montecchiani. 1-planarity testing and embedding: An experimental study. *Computational Geometry*, 108:101900, 2023. doi:10.1016/j.comgeo.2022.101900.

[8] Thomas Bläsius, Marcel Radermacher, and Ignaz Rutter. How to draw a planarization. *J. Graph Algorithms Appl.*, 23(4):653–682, 2019. doi:10.7155/JGAA.00506.

[9] Franz J. Brandenburg. Recognizing optimal 1-planar graphs in linear time. *Algorithmica*, 80(1):1–28, 2018. doi:10.1007/s00453-016-0226-8.

[10] Franz J. Brandenburg, Walter Didimo, William S. Evans, Philipp Kindermann, Giuseppe Liotta, and Fabrizio Montecchiani. Recognizing and drawing ic-planar graphs. *Theor. Comput. Sci.*, 636:1–16, 2016. doi:10.1016/J.TCS.2016.04.026.

[11] Christoph Buchheim, Dietmar Ebner, Michael Jünger, Gunnar W. Klau, Petra Mutzel, and René Weiskircher. Exact crossing minimization. In *13th International Symposium on Graph Drawing (GD' 05)*, volume 3843 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2005. doi:10.1007/11618058_4.

[12] Sergio Cabello and Bojan Mohar. Adding one edge to planar graphs makes crossing number and 1-planarity hard. *SIAM Journal on Computing*, 42(5):1803–1829, 2013. doi:10.1137/120872310.

[13] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (OGDF). In Roberto Tamassia, editor, *Handbook on Graph Drawing and Visualization*, pages 543–569. Chapman and Hall/CRC, 2013.

[14] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Christian Wolf. Inserting a vertex into a planar graph. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*, pages 375–383. SIAM, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496812.

[15] Markus Chimani, Max Ilsen, and Tilo Wiedera. Star-struck by fixed embeddings: Modern crossing number heuristics. In *29th International Symposium on Graph Drawing and Network Visualization (GD '21)*, volume 12868 of *Lecture Notes in Computer Science*, pages 41–56. Springer, 2021. URL: https://doi.org/10.1007/978-3-030-92931-2_3, doi:10.1007/978-3-030-92931-2_3.

[16] Markus Chimani, Petra Mutzel, and Immanuel M. Bomze. A new approach to exact crossing minimization. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA '08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 284–296. Springer, 2008. doi:10.1007/978-3-540-87744-8_24.

[17] Markus Chimani, Petra Mutzel, and Jens M. Schmidt. Efficient extraction of multiple kuratowski subdivisions. In *Graph Drawing, 15th International Symposium, GD 2007*, volume 4875 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2007. URL: https://doi.org/10.1007/978-3-540-77537-9_17.

[18] Markus Chimani and Mirko H. Wagner. Computing beyond-planar crossing numbers via forbidden crossing patterns. In *WALCOM: Algorithms and Computation - 20th International Conference and Workshops on Algorithms and Computation, WALCOM 2026, Perugia, Italy, March 4-6, 2026, Proceedings*, Lecture Notes in Computer Science, pages 3–18. Springer, 2026. doi:10.1007/978-981-95-7127-7_1.

[19] Bruno Courcelle. The monadic second-order theory of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 1990.

[20] Éric Colin de Verdière and Petr Hlinený. A unified FPT framework for crossing number problems. In *33rd Annual European Symposium on Algorithms, ESA 2025*, volume 351 of *LIPIcs*, pages 21:1–21:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.ESA.2025.21.

[21] Éric Colin de Verdière and Thomas Magnard. An FPT algorithm for the embeddability of graphs into two-dimensional simplicial complexes. In *29th Annual European Symposium on Algorithms, ESA 2021*, volume 204 of *LIPIcs*, pages 32:1–32:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.32.

[22] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002. doi:10.1007/S101070100263.

[23] Peter Eades, Seok-Hee Hong, Naoki Katoh, Giuseppe Liotta, Pascal Schweitzer, and Yusuke Suzuki. A linear time algorithm for testing maximal 1-planarity of graphs with a rotation system. *Theoretical Computer Science*, 513:65–76, 2013. doi:10.1016/j.tcs.2013.09.029.

[24] Michael R Garey and David S Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[25] Alexander Grigoriev and Hans L Bodlaender. Algorithms for graphs embeddable with few crossings per edge. *Algorithmica*, 49(1):1–11, 2007.

[26] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005.

[27] Thekla Hamm and Petr Hlinený. Parameterised partially-predrawn crossing number. In *38th International Symposium on Computational Geometry (SoCG '22)*, volume 224 of *LIPIcs*, pages 46:1–46:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SoCG.2022.46.

[28] Thekla Hamm, Fabian Klute, and Irene Parada. Computing crossing numbers with topological and geometric restrictions. *CoRR*, abs/2412.13092, 2024. arXiv:2412.13092.

[29] Seok-Hee Hong, Peter Eades, Naoki Katoh, Giuseppe Liotta, Pascal Schweitzer, and Yusuke Suzuki. A linear-time algorithm for testing outer-1-planarity. *Algorithmica*, 72(4):1033–1054, 2015. doi:10.1007/s00453-014-9890-8.

[30] Seok-Hee Hong and Takeshi Tokuyama, editors. *Beyond Planar Graphs*. Springer Singapore, 2020. doi:10.1007/978-981-15-6533-5.

[31] Stephen G Kobourov, Giuseppe Liotta, and Fabrizio Montecchiani. An annotated bibliography on 1-planarity. *Computer Science Review*, 25:49–67, 2017. doi:10.1016/j.cosrev.2017.06.002.

[32] Vladimir P Korzhik and Bojan Mohar. Minimal obstructions for 1-immersions and hardness of 1-planarity testing. *Journal of Graph Theory*, 72(1):30–71, 2013.

[33] Casimir Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930. URL: http://eudml.org/doc/212352.

[34] Giordano Da Lozzo and Ignaz Rutter. Planarity of streamed graphs. *Theor. Comput. Sci.*, 799:1–21, 2019. doi:10.1016/J.TCS.2019.09.029.

[35] Tomasz Łuczak, Boris Pittel, and John C Wierman. The structure of a random graph at the point of the phase transition. *Transactions of the American Mathematical Society*, 341(2):721–748, 1994.

[36] Miriam Münch, Maximilian Pfister, and Ignaz Rutter. Exact and approximate k-planarity testing for maximal graphs of small pathwidth. In *Graph-Theoretic Concepts in Computer Science - 50th International Workshop, WG 2024*, volume 14760 of *Lecture Notes in Computer Science*, pages 430–443. Springer, 2024. doi:10.1007/978-3-031-75409-8_30.

[37] Miriam Münch and Ignaz Rutter. Parameterized algorithms for beyond-planar crossing numbers. In *32nd International Symposium on Graph Drawing and Network Visualization, GD 2024*, volume 320 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.GD.2024.25.

[38] Petra Mutzel. The crossing number of graphs: Theory and computation. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 305–317. Springer, 2009. URL: https://doi.org/10.1007/978-3-642-03456-5_21.

[39] North and Rome graphs. URL: http://www.graphdrawing.org/data.html.

[40] Marc Noy, Vlady Ravelomanana, and Juanjo Rué. On the probability of planarity of a random graph near the critical point. *Proceedings of the American Mathematical Society*, 143(3):925–936, 2015.

[41] János Pach and Géza Tóth. Graphs drawn with few crossings per edge. *Combinatorica*, 17(3):427–439, 1997. doi:10.1007/3-540-62495-3_59.

[42] Sergey Pupyrev. OOPS: Optimized one-planarity solver via SAT. In *33nd International Symposium on Graph Drawing and Network Visualization, GD 2025*, volume 357 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[43] H. C. Purchase, R. F. Cohen, and M. I. James. An experimental study of the basis for graph drawing algorithms. *ACM Journal of Experimental Algorithmics*, 2:4, 1997. doi:10.1145/264216.264222.

[44] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings 5th International Symposium on Graph Drawing (GD '97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1997. doi:10.1007/3-540-63938-1_67.

[45] Helen C. Purchase, Jo-Anne Allder, and David A. 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.

[46] Marcel Radermacher, Klara Reichard, Ignaz Rutter, and Dorothea Wagner. Geometric heuristics for rectilinear crossing minimization. *ACM Journal of Experimental Algorithmics*, 24(1):1.12:1–1.12:21, 2019. doi:10.1145/3325861.

[47] Marcel Radermacher and Ignaz Rutter. Geometric crossing-minimization - A scalable randomized approach. In *27th Annual European Symposium on Algorithms (ESA'19)*, volume 144 of *LIPIcs*, pages 76:1–76:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.76.

[48] Marcel Radermacher and Ignaz Rutter. Inserting an edge into a geometric embedding. *Computational Geometry*, 102:101843, 2022. doi:10.1016/j.comgeo.2021.101843.

[49] Marcus Schaefer. Picking planar edges; or, drawing a graph with a planar subgraph. In *Graph Drawing - 22nd International Symposium, GD 2014*, volume 8871 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2014. URL: https://doi.org/10.1007/978-3-662-45803-7_2.

[50] Marcus Schaefer. The graph crossing number and its variants: A survey. *The Electronic Journal of Combinatorics*, pages DS21–May, 2021. doi:10.37236/2713.

[51] Yusuke Suzuki. 1-planar graphs. In *Beyond Planar Graphs: Communications of NII Shonan Meetings*, pages 47–68. Springer, 2020. doi:10.1007/978-981-15-6533-5_4.