Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 27, no. 2, pp. 71–94 (2023)
DOI: 10.7155/jgaa.00615

Star-Struck by Fixed Embeddings: Modern Crossing Number Heuristics

Markus Chimani *Max Ilsen* *Tilo Wiedera*

Theoretical Computer Science, Osnabrück University, Osnabrück, Germany

Submitted: November 2021	Reviewed: March 2022	Revised: April 2022
Accepted: October 2022	Final: October 2022	Published: February 2023
Article type: Regular Paper	Communicated by: I. Rutter, H. Purchase	

Abstract. We present a thorough experimental evaluation of several crossing minimization heuristics that are based on the construction and iterative improvement of a planarization, i.e., a planar representation of a graph with crossings replaced by dummy vertices. The evaluated heuristics include variations and combinations of the well-known planarization method, the recently implemented star reinsertion method, and a new approach proposed herein: the mixed insertion method. Our experiments reveal the importance of several implementation details such as the detection of non-simple crossings (i.e., crossings between adjacent edges, multiple crossings between the same two edges, or crossings of an edge with itself). The most notable finding, however, is that the insertion of stars in a fixed embedding setting is not only significantly faster than the insertion of edges in a variable embedding setting, but also leads to solutions of higher quality.

1 Introduction

Given a graph G, the *crossing number* problem asks for the minimum number of edge crossings in any drawing of G, denoted by $cr(G)$. This problem is NP-complete [19], even when G is restricted to cubic graphs [23] or graphs that become planar after removing a single edge [7]. While the currently known integer linear programming approaches to the problem [6, 15, 16] solve sparse instances within a reasonable time frame [12], dense instances require the use of heuristics.

One such heuristic is the well-known *planarization method* [1, 21], which constructs a *planarization*, i.e., a planar representation of G with crossings replaced by dummy vertices of degree 4. The heuristic first computes a spanning planar subgraph of G and then iteratively inserts the

Special Issue on the 29th Int. Symposium on Graph Drawing and Network Visualization, GD 2021

Supported by the German Research Foundation (DFG) grant CH 897/2-2.

E-mail addresses: markus.chimani@uos.de (Markus Chimani) max.ilsen@uos.de (Max Ilsen) tilo.wiedera@uos.de (Tilo Wiedera)

remaining edges. Several variants of the planarization method have been thoroughly evaluated, including different edge insertion algorithms and postprocessing strategies; see [10] for the latest study. In a recent paper [17], Clancy et al. present an alternative heuristic—the *star reinsertion method*—, which differs in two key aspects from the planarization method: It (i) starts with a full planarization (instead of a planar subgraph) that is iteratively improved by reinserting elements, and (ii) the reinserted elements are stars (vertices with their incident edges) rather than individual edges. These star insertions are performed using a straight-forward but never tried algorithm from literature [13]. Clancy et al. were faced with the problem that the implementations of the aforementioned heuristics were written in different languages, leading to incomparable running times. In their evaluation, they thus focus on variants of the star reinsertion method; their comparison with the planarization method only gives averages over (a quite limited number of) full instance sets and relies on old data from previous experiments.

Herein, we present a comprehensive experimental evaluation of a wide array of crossing minimization heuristics based on edge and star insertion encompassing all known strong candidates. This includes not only variants of the planarization and star reinsertion methods but also *combined* approaches. In addition, we present and evaluate a new heuristic that builds up a planarization from a planar subgraph using *both* star and edge insertions. All of these algorithms are implemented as part of the same framework, enabling us to accurately compare their running times. Furthermore, we suggest ways of simplifying the implementation of the heuristics, increasing their speed in practice, and improving their results—e.g., by properly handling crossings between adjacent edges and multiple crossings between the same two edges.

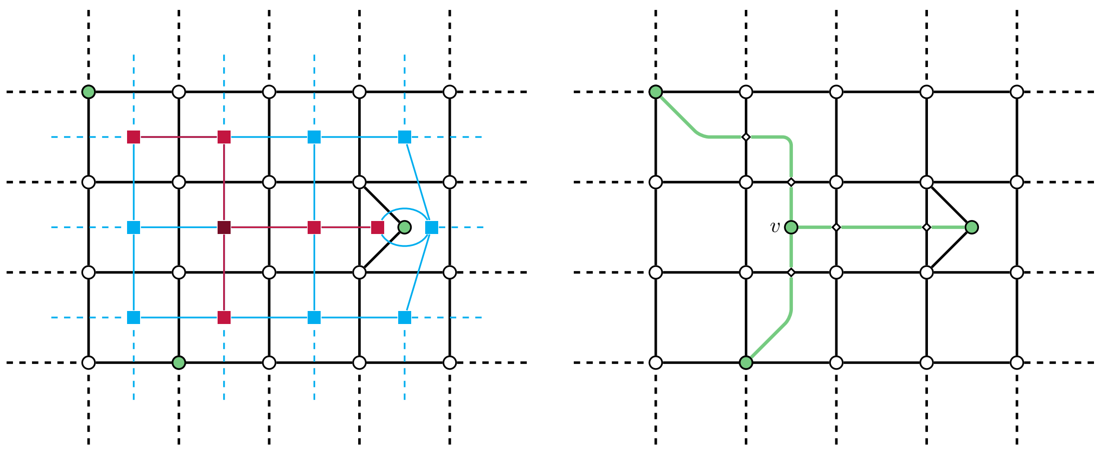
The goals of this evaluation are to indicate which algorithms are most effective in which scenarios and to discern those approaches that are generally worth implementing and developing further in future research. This requires a comparison of the solution quality and running time of the algorithms when all of them are implemented and run under the same conditions. It is explicitly not our goal to find the overall fastest *implementation* of any crossing minimization heuristic. E.g., the implementation in [17] is `C` code dealing with immutable graphs and incompatible with the implementation in [10], which is part of a framework for dynamically changeable graphs. Hence, we report on a faithful reimplementation of the former (with the same solution qualities and running time guarantees) within the latter framework.

This paper is structured as follows: First, Section 2 introduces the basic graph terminology and notation. Section 3 describes the evaluated algorithms in detail, starting with the basic algorithms used to solve insertion problems and continuing with the crossing minimization heuristics themselves. Further, we define non-simple crossings and note on their significance in Section 4. The setup and the results of our experimental evaluation are described in Section 5, with Section 6 summarizing our findings.

2 Preliminaries

In the following, we consider a connected undirected graph G (that is usually simple, i.e., does not contain parallel edges or self-loops) with n vertices and m edges, denoted by $V(G)$ and $E(G)$ respectively. Let Δ be the maximum degree of any vertex in $V(G)$ and $N(v) := \{w \mid (v,w) \in E\}$ the neighborhood of a vertex v. Then, v along with a subset of its incident edges $F \subseteq \{(v,w) \in E\}$ is collectively called a *star*, denoted by (v, F).

We define a *drawing* of a graph G to be a mapping of its vertices to points in the Euclidean plane as well as a mapping of its edges to curves whose endpoints are the points of their incident vertices. An edge curve may not intersect with any other vertex point. Furthermore, two edge



(a) Graph and its dual with insertion spider **(b)** Graph with inserted star (v, F)

Figure 1. Insertion of a star (v, F) into a graph G (drawn in black); the edges in F shall connect v with the three green vertices. On the left, we can see (a part of) G prior to the insertion of (v, F)—its dual G^* is drawn in cyan and red with square vertices. The red dual vertices correspond to faces of an insertion spider. This insertion spider consists of three insertion paths, one for each edge in F. The darker red vertex marks the common face that all insertion paths share. On the right, we can see the graph after the insertion of (v, F) in accordance with the red insertion spider. Note that the insertion is optimal in a fixed embedding setting since at least five crossings will be created when (v, F) is inserted into the given embedding. In a variable embedding setting, however, it is possible to move the rightmost green vertex one face to the left and then insert (v, F) into G with just four resulting crossings.

curves can only have a finite number of points in common, and no three edge curves intersect in a common non-endpoint. We call common non-vertex points between two edge curves *crossings*. If it does not introduce ambiguity, we may use the term *edge* when referring to a curve in a drawing. A drawing is *simple* if and only if each pair of edges intersects at most once (either in a crossing or a common endpoint) and no edge intersects itself. Given a drawing of a graph G, a *planarization* is a planar graph H obtained from this drawing where crossings are replaced by degree-4 vertices.

A (combinatorial) *embedding* of a planar graph G corresponds to a cyclic order of the edges around each vertex in $V(G)$ such that a drawing respecting this ordering can be realized without any edge crossings. This induces a set of cycles that bound the *faces* of the embedding. Based on an embedding of the *primal graph* G, we can define the *dual graph* G^*, whose vertices correspond to the faces of G, and vice versa. For each primal edge $e \in E(G)$, there exists a dual edge $e^* \in E(G^*)$ between the dual vertices corresponding to the e-incident primal faces (see Figure 1a, which displays a graph and its dual). Note that G^* may be a multi-graph with self-loops even if G is simple.

For the purpose of this paper, it is of particular concern how to insert an edge (v_1, v_2) into a planarization. First, it is necessary to find a corresponding *insertion path*, i.e., a sequence of faces $f_1, \ldots, f_k$ such that v_1 is incident to f_1, v_2 incident to f_k, and f_i adjacent to f_{i+1} for $i \in \{1, \ldots, k-1\}$. An edge between v_1 and v_2 can then be inserted into a planarization by subdividing a common edge for each face pair (f_i, f_{i+1}) and routing the new edge as a sequence of edges from v_1 along the subdivision vertices to v_2. By extension, an *insertion spider* of a star (v, F) is a set of insertion paths, one for each edge in F. These insertion paths necessarily share a common face into which v can be inserted. Figure 1 visualizes an insertion spider and the corresponding star insertion.

3 Algorithms

The subject of our experimental evaluation is a wide array of crossing minimization heuristics. Each heuristic makes use of at least one edge or star insertion algorithm as a subroutine. Hence, these different insertion algorithms are discussed first in the following section. The subsequent section then explains the overall crossing minimization heuristics in detail.

3.1 Solving Insertion Problems

Insertion problems, and their efficient solutions, form the cornerstone of all known strong crossing minimization heuristics. We first consider the insertion of edges and stars in a setting where the given embedding is fixed. Afterwards, we discuss the variable embedding setting, in which an optimal embedding has to be found by the algorithm.

Definition 1 (EIF, SIF) *Given a planar graph G, an embedding Π of G, and an edge (or star) not yet in G, insert this edge (star) into Π such that the number of crossings in Π is minimized. We refer to these problems as the* edge (star) insertion problem with fixed embedding EIF (SIF, resp.).

Given a primal vertex v, let $\hat{v}$ be the vertex that is created by contracting the dual vertices that correspond to v-incident faces. Then, the EIF for any given edge (v_1, v_2) can be solved optimally in $\mathcal{O}(n)$ time by computing the shortest path from $\hat{v_1}$ to $\hat{v_2}$ in the dual graph G^* via breadth-first search (BFS) [1]. By extension, the SIF for a star (v, F) can be solved in $\mathcal{O}(|F| \cdot n)$ time as follows [13]: For each edge $(v, w) \in F$, solve the single-source shortest path problem in G^* with $\hat{w}$ as the source (via BFS). For each face f, the sum over all of the resulting distance values at this f then represents the number of crossings that would be created if v was to be inserted into f. Hence, the face with the minimum distance sum is the optimal face to insert v into, and the computed shortest paths to this face collectively form an insertion spider. To avoid crossings between these shortest paths (due to them not being necessarily unique), we can construct an insertion spider using a final BFS starting at the optimal face. Figure 1 shows the optimal insertion of a star in a fixed embedding setting (and the corresponding insertion spider).

Definition 2 (EIV, MEIV, SIV) *Given a planar graph G and an edge (a set of k edges, or a star) not yet in G, find an embedding Π among all possible embeddings of G such that optimally inserting the edge (set of k edges, star) into this Π results in the minimum number of crossings. We refer to these problems as the* edge (multiple edge, star) insertion problem with variable embedding EIV (MEIV, SIV, resp.).

The EIV can be solved in $\mathcal{O}(n)$ time using an algorithm by Gutwenger et al. [22], which finds a suitable embedding (with the help of SPR-trees) and then executes the EIF-algorithm described above. Now consider the MEIV: Solving it for general k is NP-hard [28], however there exists an $\mathcal{O}(kn + k^2)$-time approximation algorithm with an additive guarantee of $\Delta k \log k + \binom{k}{2}$ [14] that performs well in practice [10]. Put briefly, the EIV-algorithm is run for each of the k edges independently, and a single final embedding is identified by combining the individual (potentially conflicting) solutions via voting. Then, the EIF-algorithm can be executed once for each edge. Note that the SIV can be solved optimally in polynomial time by using dynamic programming techniques [13]. However, for graphs that are not series-parallel, the resulting running times are exorbitant and there is no known implementation of this algorithm. In fact, our results herein suggest that in the context of crossing minimization heuristics, the solution power of the SIV-algorithm is fortunately not necessary in practice.

Each problem discussed above has a *weighted* version which can be solved in the same manner if each c_e-weighted edge e is replaced by c_e parallel 1-weighted edges beforehand [9,26]. In practice it is worthwhile to compute the shortest paths during the EIF/SIV-algorithm on the weighted instance directly. However, this does not allow for the same theoretical upper bounds of the running times since the weights may be arbitrarily large.

3.2 Crossing Minimization Heuristics

In this section, we review the crossing minimization heuristics that are to be evaluated in Section 5. Each of the heuristics uses one of three main algorithms that iteratively build up a planarization, starting with a planar subgraph. Some of these have exchangeable subroutines which are described below. Furthermore, we evaluate the algorithms not only on their own but also in combination with the star reinsertion method, a postprocessing strategy proposed by Clancy et al. [17]. In particular, our selection includes all of the strongest algorithms from the most recent experimental crossing minimization heuristic studies [10,17], as well as a new algorithm class we propose herein.

Each algorithm and subroutine is given its own abbreviation that will be used in the evaluation in Section 5. To refer to specific algorithm variants (with designated subroutines) or multiple algorithms executed in series, the abbreviations are concatenated as described in Figure 2. Figure 2 can also be used as a flow chart to understand the different algorithms and subroutines that each crossing minimization heuristic is composed of.

The planarization method (*plm*) is the longest studied and best-known approach considered, achieving strong results in previous evaluations [1,10,21]. First, we compute a spanning planar subgraph $G' = (V, E') \subseteq G$, usually by employing a maximum planar subgraph heuristic and extending the result such that it becomes (inclusion-wise) maximal. Then, the remaining edges $F := E \setminus E'$ are either inserted one after another—by solving the respective EIF (*fix*) or EIV (*var*)—or simultaneously using the MEIV-approximation algorithm (*multi*).[1] Gutwenger and Mutzel [21] describe a postprocessing strategy for *plm* based on edge insertion: Each edge is deleted from the planarization and reinserted one after another (*all*). To incrementally improve the planarization, *all* can also be executed once after each individual edge insertion (*inc*) [10]. When neither *all* nor *inc* is employed, we use the specifier *none* instead.

The chordless cycle method (*ccm*) realizes the idea of extending a *vertex-induced* planar subgraph to a full planarization via star insertion [13]. It is the best-performing scheme for the star insertion algorithm as examined by Clancy et al. [17]: Search for a chordless cycle in G, e.g., via breadth-first search. Let G' denote the subgraph of G that is already embedded and initialize it with this chordless cycle. Iteratively (until the whole graph is embedded) select a vertex $v \notin V(G')$ such that there exists at least one edge (v, w) that connects v with the already embedded subgraph G'; insert v into G' by solving the SIF for the star $(v,\ \{(v, w) \in E \mid w \in V(G')\})$.

[1]We use *fix*, *var*, and *multi* as abbreviations instead of *EIF*, *EIV* and *MEIV* in order to stay consistent both with the implementation code and with previous papers on the planarization method [10,21].

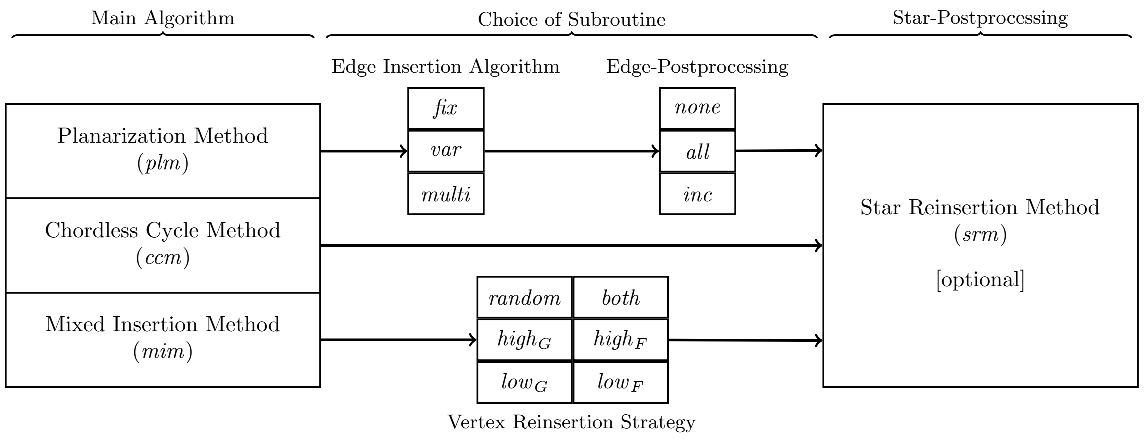


Figure 2. Visualization of the evaluated crossing minimization heuristics and their variants. Starting with a base algorithm—*plm*, *ccm*, or *mim*—, one has to specify which subroutine variants it should use. For *plm*, this includes the edge insertion algorithm and the edge-postprocessing strategy; for *mim*, the vertex reinsertion strategy. At the end, one can choose to employ *srm* for further postprocessing. The shorthand naming scheme for the evaluated crossing minimization heuristics also follows this flowchart: We simply concatenate the abbreviations of the main algorithm, subroutines, and (if used) the star-insertion-based postprocessing, e.g. *mim-both-srm*. We may omit prefixes if they are clear from the context, e.g., we may write *fix-all* instead of *plm-fix-all*. The 2012 experimental study [10] considered all variants *plm*-{*fix*,*var*,*multi*}-{*none*,*all*,*inc*}. The 2019 study [17] considered *ccm-srm* together with simple alternatives to *ccm* that are not listed here as they did not perform as well as *ccm-srm*.

The mixed insertion method (*mim*) is a novel approach that we propose as an alternative to the planarization schemes above. It proceeds in a fashion that is similar to *plm* but relies on star insertion instead of edge insertion in as many cases as possible. Accordingly, let G' denote the subgraph of G that is already embedded and initialize it with a spanning planar subgraph $(V, E') \subseteq G$. Then, (attempt to) insert the remaining edges $F := E \setminus E'$ by reinserting at least one endpoint of each edge $e \in F$ via star insertion. Since removing and then reinserting a *cut vertex* of the planar subgraph G' would temporarily disconnect it, the cut vertices of the planar subgraph are computed (cf. [24]) and each edge $e \in F$ is processed as follows: If both endpoints of e are cut vertices of G', insert the edge via edge insertion (we choose to do so in a variable embedding setting as such edge insertions happen rarely). If only one endpoint of the edge is a cut vertex, reinsert the other one. If neither endpoint of the edge is a cut vertex, the endpoint to be reinserted can be chosen freely—globally, this corresponds to finding a vertex cover on the graph induced by F that has to include all vertices neighboring a cut vertex in G'. Finding a minimum vertex cover is NP-hard [25]; therefore we compare several heuristics: For each edge e, choose one of the endpoints randomly (*random*), choose the one with the higher or lower degree in G ($high_G$, low_G), choose the one with the higher or lower degree in the graph induced by all edges in F not incident to a cut vertex in G' ($high_F$, low_F), or choose both endpoints (*both*). Each of the chosen vertices is then deleted from the planar subgraph and reinserted together with all of its edges in the original graph by solving the corresponding SIF.

The *star reinsertion method* (*srm*) is a star-insertion-based postprocessing strategy proposed by Clancy et al. [17]. It starts with an already existing planarization, which may be constructed using any of the methods outlined above (or even more trivial ones, such as extracting a planarization from a circular layout of the vertices, which, however, is known to perform worse [17]). The given planarization is then processed as follows: Iteratively choose a vertex v, delete v from G, and reinsert it again by solving the SIF for the star $(v, v \times N(v))$. Continue the loop until there is no further vertex whose reinsertion improves the solution (in which case the latter is said to be *locally optimal*). Clancy et al. propose different methods for choosing v; here, we consider the scheme they report to be the best compromise between solution quality and running time: In each iteration, try to reinsert every vertex once and continue with the next iteration as soon as a vertex is found whose reinsertion improves the number of crossings in the planarization.

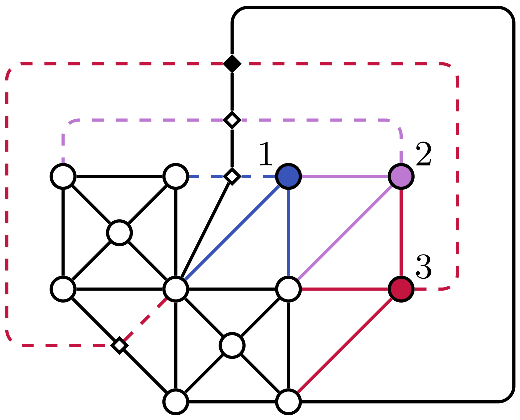
The original algorithm only updates a planarization once an actual improvement is found and resets it to its original state otherwise. We propose to never reset it. This approach is permissible as the SIF is solved optimally and the number of crossings hence never increases after the reinsertion of a star. Not resetting the planarization saves time in practice as it allows for a simpler implementation without any need to copy the dual graph, and at the same time is expected (in the statistical sense) to yield solutions of the very same quality.

4 A Note on Non-simple Crossings

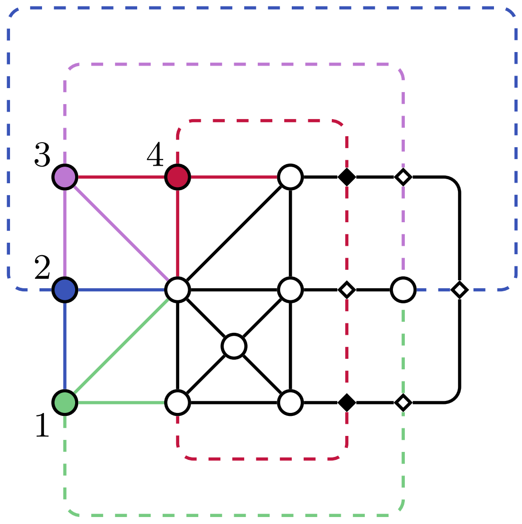
It is well-known that any crossing-optimal drawing can be assumed to be *simple*, i.e., it may not contain crossings between adjacent edges (α*-crossings*), multiple crossings between a pair of edges (β*-crossings*), or crossings of an edge with itself (γ*-crossings*). We call any such undesired crossings *non-simple*. Surprisingly, earlier implementations of the planarization method did not consider the emergence and removal of any non-simple crossings [10], while the implementation of the star reinsertion method by Clancy et al. only considers β- but neither α- nor γ-crossings [17]. However, we show in Figure 3 that incrementally solving the same kind of insertion problem may result in a planarization with α- or β-crossings, even when starting with a planar subgraph. These α- and β-crossings can be removed by reassigning edges in the planarization to different edges in the original graph and then deleting the respective dummy vertices. As Figure 4 makes evident, such a removal can turn other crossings into non-simple ones as well—hence, the touched edges will have to be checked for new non-simple crossings afterwards. However, the removal procedure will end eventually as the planarization contains finitely many crossings and at least one of them is removed in each non-final iteration. Note that during the iterations, γ-crossings may also be produced, as is shown in Figure 5. Each of these can in turn be removed by deleting the respective cycle (as well as all crossings on this cycle) starting and ending at the γ-crossing. Removing non-simple crossings leads to significantly better results overall, see Section 5.4.

5 Experiments

In the following, we present our experiments and their results. First, we describe the experimental setup—i.e., the code and the environment it is compiled and executed in—as well as the instances—i.e., which graph sets we are using and how these graphs are preprocessed. The subsequent sections then compare the running times and results of different groups of algorithms. Section 5.1 begins by discussing *fast* heuristics that have the drawback of producing a large number of crossings: *mim*, *ccm*, and *fix-none*. All the different variants of *plm*, the planarization method, are then

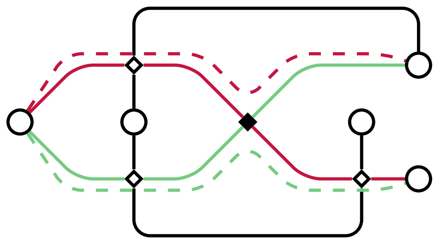


(a) Creation of an α-crossing

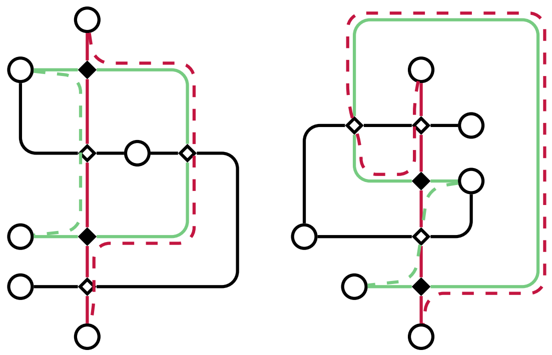


(b) Creation of a β-crossing

Figure 3. A non-simple crossing on the red dashed edge as the result of incrementally solving the same kind of insertion problem. When starting with the black planar subgraph, this may happen by solving the SIV using the described algorithm for the colored vertices in the order of their label numbers. Alternatively, if all solid edges constitute the initial planar subgraph, solving the EIV for the dashed edges in the order of their label numbers can have the same result. The examples apply both in the fixed and the variable embedding setting. Dummy vertices for crossings are represented by small diamonds; the diamonds are black if the crossings are non-simple.



(a) Removal of an α-crossing



(b) Removal of a β-crossing

Figure 4. Non-simple crossings (of type α and β respectively) between the red and green edges. After their removal (new edge paths drawn as dashed), the red edge is involved in a new non-simple crossing of the same type and the green edge in a new non-simple crossing of the opposite type. Thus, the removal procedure may have to be iterated.

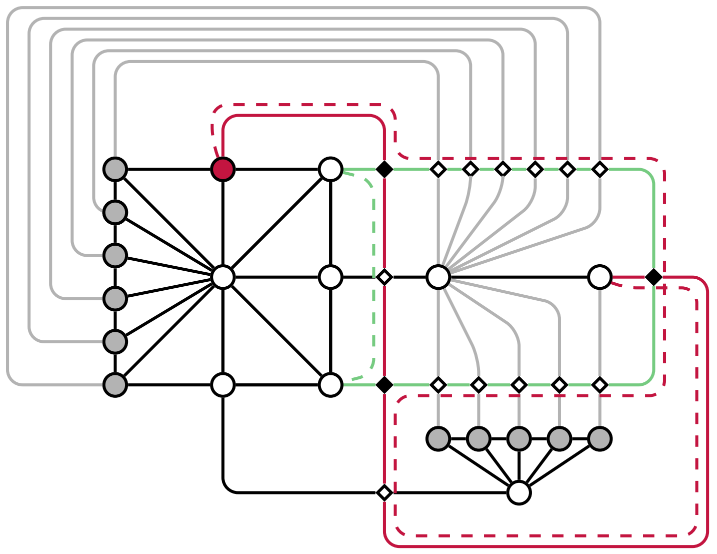


Figure 5. Creation of a γ-crossing on the red edge as a result of removing the β-crossing between the red and the green edge (edge paths after the removal are drawn as dashed). In order to create a γ-crossing in this way, the red and the green edge are required to cross each other three times, which may occur when proceeding as follows: Start with the planar subgraph consisting of black and green edges, insert the gray edges in any order, and lastly insert the red edge. Alternatively, start with the planar subgraph that consists of all black and green edges not incident to a gray or red vertex, insert the gray vertices with their respective incident edges in any order, and finally insert the red vertex with its incident edges. The example applies both in the fixed and the variable embedding setting.

examined and compared in Section 5.2. Once these base algorithms and their variants have been discussed, Section 5.3 takes a look at the improvements achieved by *srm*, i.e., the star-insertion-based postprocessing. Finally, Sections 5.4 and 5.5 investigate the effectiveness of removing non-simple crossings and running multiple permutations of the same algorithm respectively. At the end of each section, we summarize the main observations made in that section.

Setup. All algorithms are implemented in `C++`. The implementations are publicly available as part of the Open Graph Drawing Framework (OGDF [11], `www.ogdf.net`), release “2022.02 Dogwood”. The code is compiled with GCC 8.3.0 using optimization level `-O3`. Each computation is performed on a single physical processor of a Xeon Gold 6134 CPU (3.2 GHz) running Debian GNU/Linux 10 “Buster” with Linux kernel version 4.19.0-6-amd64. We do not enforce a time limit but a memory limit of 4 GB, which only affects memory-intensive algorithms on instances with an extremely large number of crossings. Computations reaching this memory limit already last several hours without any prospect of terminating within a reasonable time frame. Whenever the memory limit takes effect, we mention this in the evaluation.

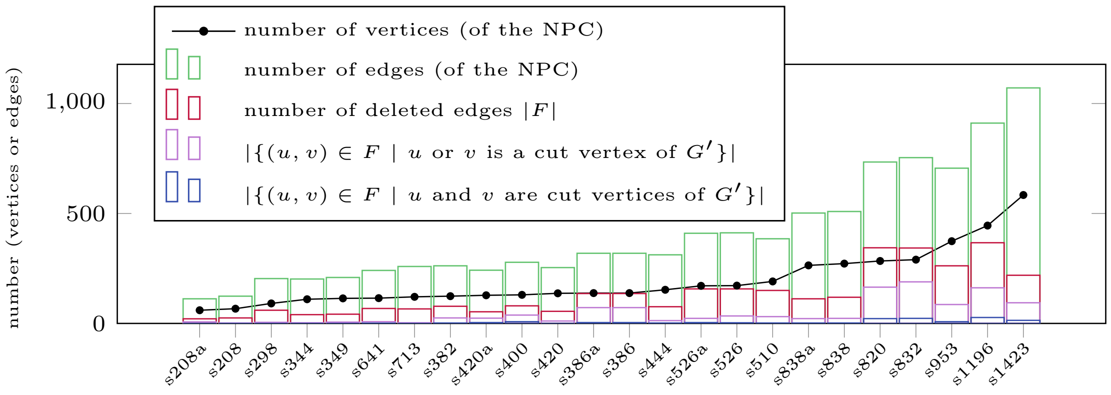
All instances and results are available for download at `https://tcs.uos.de/research/cr`. This webpage also includes instructions on where to download the code and how to run all of the heuristics examined herein. Furthermore, one can find a document with additional plots for all algorithms and instance sets there.

Instances. Table 1 lists the instance sets used for our evaluation, and Figures 6 and 7 give an overview of their size distribution, including the number of edges that are deleted to create the planar subgraph. The plots also display the number of these deleted edges that are incident to one or two cut vertices of the planar subgraph as these are of particular relevance for the mixed insertion method (see Section 3.2). In the plots, n is rounded up to the nearest multiple of five or—in the case of Figure 6e—ten. The densities in Figure 7c are rounded up to the nearest increment of 0.03, starting at 0.5.

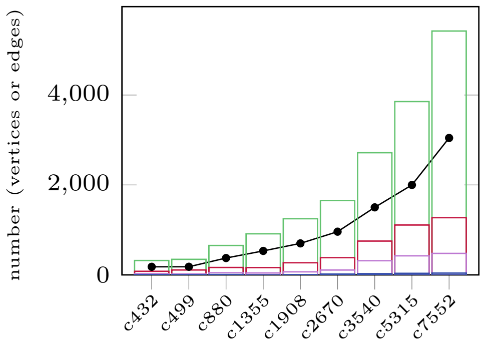
To enable a proper comparison of the tested algorithms (and potentially in the future, their competitors), we consider multiple well-known benchmark sets as well as constructed, random, and real-world instances with varying characteristics. These are preprocessed by computing the *non-planar core* (NPC) [9] for each non-planar biconnected component. We consider only those instances that have at least 25 vertices after the NPC reduction unless the instance is part of the Complete, Complete-Bip., or KnownCR instance sets. Moreover, we precompute a planar subgraph and chordless cycle for each instance such that different runs of *plm*, *mim*, and *ccm* can be started with the same initialization. The planar subgraph is computed by using Chalermsook and Schmid's diamond algorithm [8] and extending the result to a maximal planar subgraph. On average, this computation took only 0.77% of the time needed to execute the fastest evaluated heuristic *fix-*

Table 1. Considered instance sets. "#" denotes the number of graphs and $|V(G)|$ the (range of the) numbers of vertices—both values refer to the instance sets *after* preprocessing. Further, let $\square$ denote the Cartesian product of two graphs, C_i the cycle with i edges, P_j the path with j edges, and G_k the 21 non-isomorphic connected graphs on 5 vertices indexed by k.

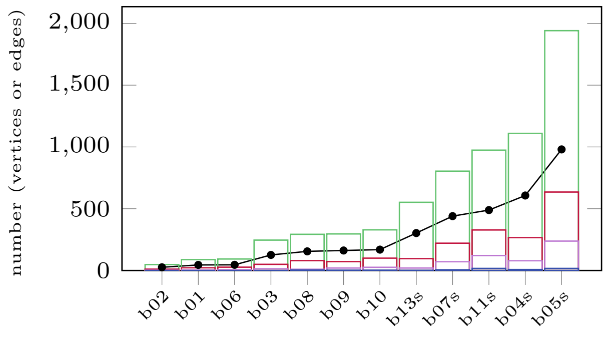
Name	#	$\|V(G)\|$	Description
Rome	3668	25–58	Well-known benchmark set [3] of originally 11,528 planar and non-planar graphs with 10–100 vertices, sparse
North	106	25–64	Well-known benchmark set collected by S. North at AT&T Bell Labs via the e-mail graph drawing service "Draw DAG" [2]
Webcompute	75	25–112	Instances sent to our online tool [16] for the exact computation of crossing numbers, `crossings.uos.de`
Expanders	240	30–100	20 random regular graphs [27] (*expander graphs* with high probability) with node degree δ for each parameterization $(\|V(G)\|, \delta) \in \{30, 50, 100\} \times \{4, 6, 10, 20\}$
Circuit-Based	45	26–3045	Hypergraphs from real world electrical networks, transformed into traditional graphs by replacing each hyperedge h by a new hypervertex connected to all vertices contained in h
ISCAS-85 [5]	9	180–3045	
ISCAS-89 [4]	24	60–584	
ITC-99 [18]	12	26–980	
KnownCR	1946	9–250	Benchmark set with *cr* known through proofs [20]:
$C \square C$	251	9–250	$\to C_i \square C_j$ with $3 \le i \le 7$, $j \ge i$ such that $i \cdot j \le 250$
$G \square P$	893	15–245	$\to$ Subset of $G_i \square P_j$ with $1 \le i \le 21$, $3 \le j \le 49$
$G \square C$	624	15–250	$\to$ Subset of $G_i \square C_j$ with $1 \le i \le 21$, $3 \le j \le 50$
$P(\_,\_)$	178	10–250	$\to$ Generalized Petersen graphs $P(2k+1, 2)$ with $2 \le k \le 62$ and $P(m, 3)$ with $9 \le m \le 125$
Complete	46	5–50	Complete graphs K_n for $5 \le n \le 50$
Complete-Bip.	666	10–80	Complete bipartite graphs K_{n_1,n_2} for $5 \le n_1, n_2 \le 40$



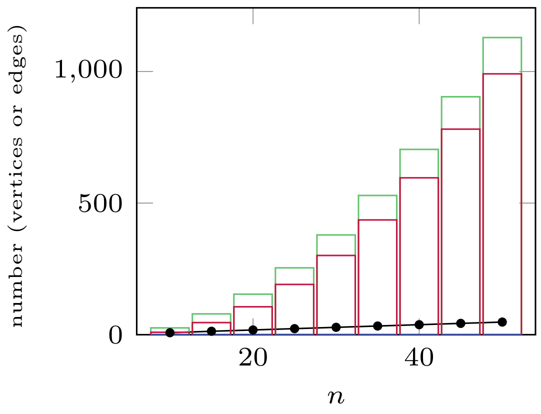
(a) ISCAS-89



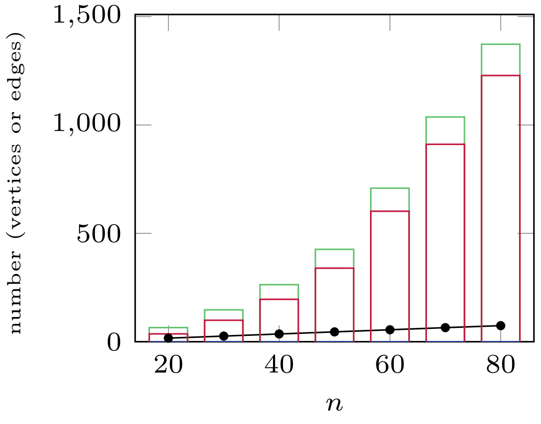
(b) ISCAS-85



(c) ITC-99

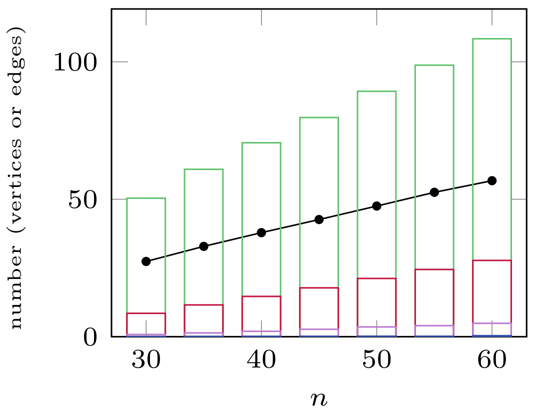


(d) Complete

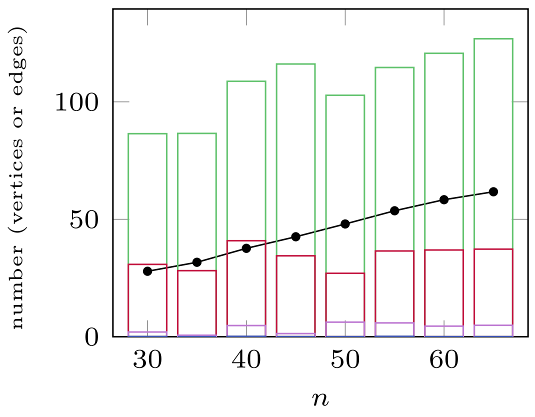


(e) Complete-Bip.

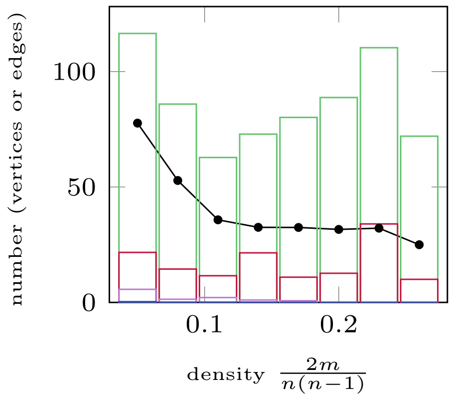
Figure 6. Statistics on circuit-based and complete (bipartite) instances.



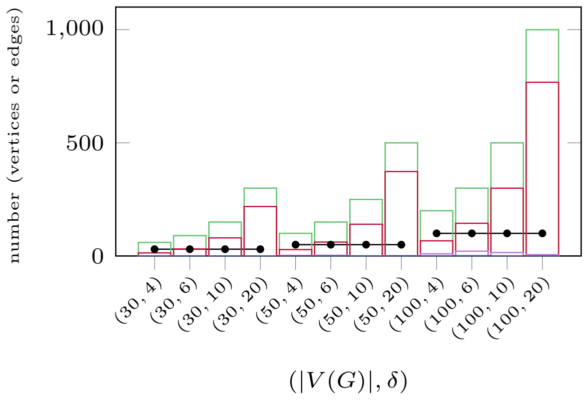
(a) Rome



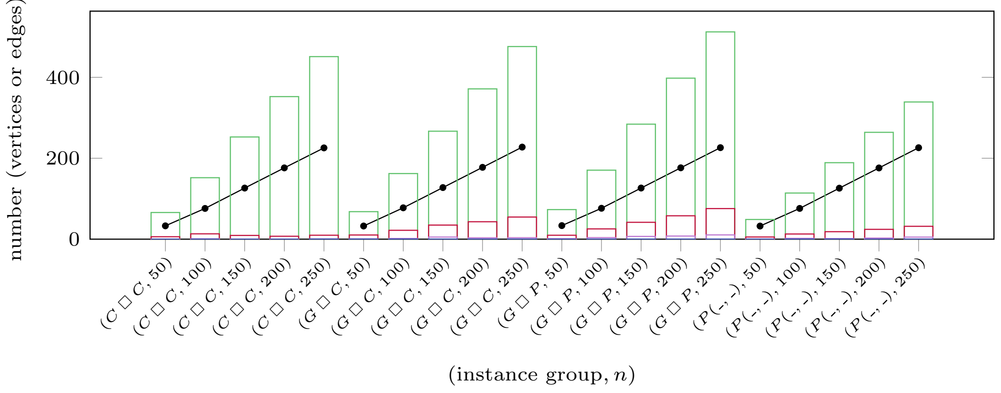
(b) North



(c) Webcompute



(d) Expanders



(e) KnownCR

Figure 7. Statistics on the other instances.

none—a comparatively negligible amount of time that is not further considered in the evaluation.

The precomputed chordless cycle almost always consists of 3–6 vertices, containing 7–11 vertices for only 15 instances overall. How many edges are deleted to create the planar subgraph, on the other hand, varies greatly depending on the size and density of the graph. Of particular interest is the number of deleted edges that are incident to one or two cut vertices of the planar subgraph: During *mim*, the former ones have a fixed endpoint that must be reinserted via star insertion (disallowing a choice of the reinserted endpoint) while the latter ones must be inserted via edge insertion. Clearly, more dense instances such as the complete (bipartite) ones and the expanders require more edges to be deleted to form a planar subgraph. At the same time, due to their high connectivity, these instances also have fewer deleted edges that are connected to cut vertices in the planar subgraph. In particular, the complete (bipartite) instances do not have a single such edge. However, even on the sparser instances, *mim* inserts almost all edges via star insertion and one can usually choose the endpoint to be reinserted (see the *mim*-variants described in Section 3.2).

5.1 Fast Heuristics: Mixed Insertion Method, Chordless Cycle Method and Fixed Embedding Edge Insertion

To begin with, we compare those heuristics that are very fast but yield a comparably high number of crossings. This includes the *mim*-variants, *ccm*, and *fix-none* (all without *srm*-postprocessing). We first aim to find the most promising *mim*-variant, and then compare this variant with *ccm* and *fix-none* in terms of quality and running time. Figures 8 and 9 display some representative results on the Rome graphs and the expanders, contrasting these heuristics with *BEST*, which denotes the best solution found by 50 random permutations of any heuristic tested herein (cf. Section 5.5).

Among the *mim*-variants, there are only little differences in computation speed and resulting number of crossings. However, reinserting *both* endpoints whenever a choice between two endpoints can be made clearly provides the best results across all instances while only taking an insignificant amount of additional time. The variant leads to the highest amount of reinserted stars and hence also to more chances for an improvement of the number of crossings. In fact, for Rome and North instances, the second best (but also second slowest) variant is the one that reinserts a *random* endpoint, leading to the conclusion that the variants other than *both* should generally not be considered if one aims for a high solution quality. If a low running time is preferred, however, one might use $high_F$ as it needs the lowest amount of star insertions and is thus the fastest variant (but provides results of mixed quality). Here, reinserting the endpoints with a higher degree pays off: The low_F variant is in comparison to $high_F$ not only slower, but always produces subpar results (e.g., on KnownCR instances, it consistently has the worst performance among all *mim*-variants).

Compared with *fix-none* and *ccm*, *mim* (from now on always referring to the *both*-variant) provides better results on almost all instances. The fastest of the algorithms, on the other hand, is *fix-none*. The last of the three, *ccm*, should only be considered when examining particularly dense instances: On the sparse instance set Rome (KnownCR), it is 1.8 (5.4) times slower and yields 1.68 (4.08) times more crossings relative to BEST in comparison with *fix-none* (which in turn yields worse results than *mim*). The solution and speed disparity between the algorithms becomes smaller on instances with a higher density—see, e.g., Figure 9. On complete (bipartite) instances, *ccm* even surpasses *mim* both in terms of solution quality and speed.

Key takeaways. The *mim*-variant with the best results quality-wise is *both*; and its running time is not much worse than that of the fastest *mim*-variant $high_F$. It produces fewer crossings than *ccm* and *fix-none*. In contrast, *fix-none* is the fastest of all heuristics tested herein, having the lowest running time on 55.29% of all instances.

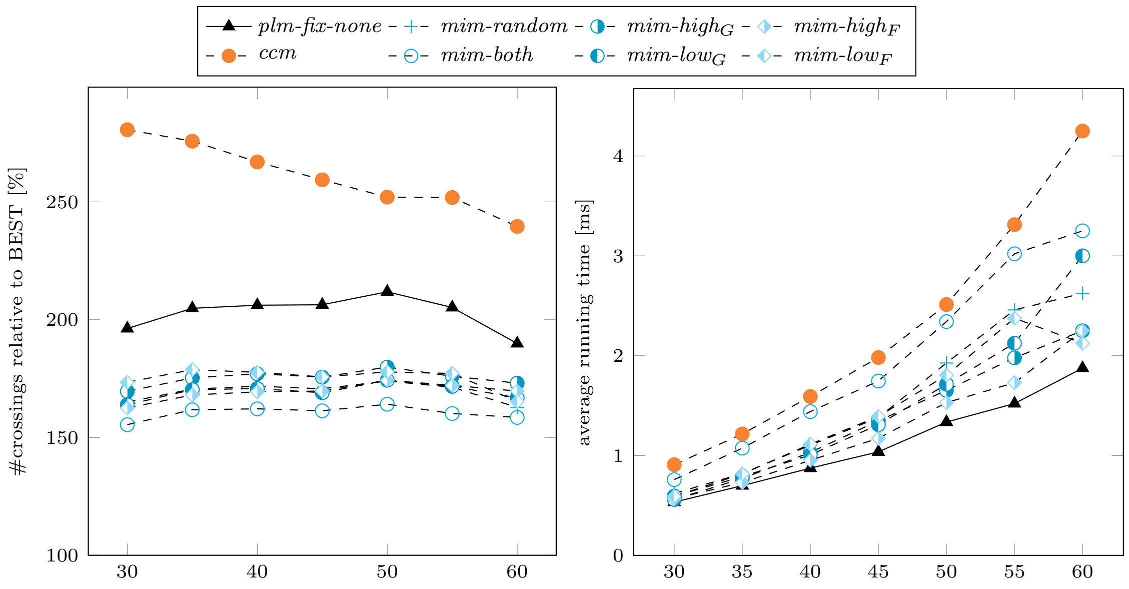


Figure 8. Comparison of the *mim*-variants, *ccm*, and *fix-none* on the Rome graphs.

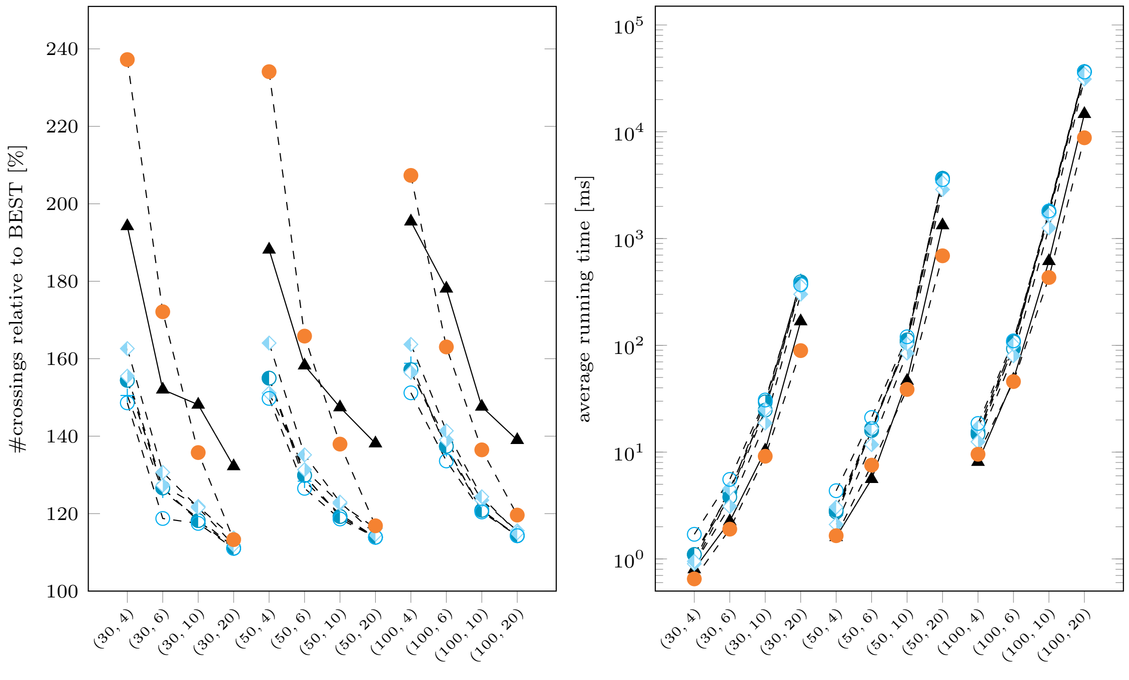


Figure 9. Comparison of the *mim*-variants, *ccm*, and *fix-none* on the expanders.

5.2 Planarization Method

The different edge insertion algorithms and edge-postprocessing strategies for the planarization method allow to greatly improve the final planarizations at the cost of additional running time. Hence, this section investigates which *plm*-variants produce the fewest crossings, which are the fastest, and which form a good compromise between both of these objectives. We first evaluate the edge insertion algorithms and postprocessing strategies and then discusses how to reach a good compromise between speed and solution quality. A detailed experimental comparison of these *plm*-variants was already carried out in 2012 [10]. We are able to replicate the results of that study and corroborate its claims with our findings on the additional instances considered in this paper. For example, Figure 10 showcases the results and running times on the ISCAS-89 instances, on which *plm* has not been evaluated before.

Regarding edge insertion algorithms, the value of crossings relative to BEST for *var* is on average 4.78 percentage points lower than for *multi* (but *var* is also 20.2 times slower). In turn, *multi* performs better than *fix*. This hierarchy is especially evident (across all instance sets) when no edge-postprocessing is employed, but on the Rome instances it also persists for *all* and for *inc*. The differences in quality between solutions produced with and without edge-postprocessing are much larger than for different edge insertion algorithms: *none* provides much worse results than *all* and *inc* across all instance sets. However, the latter two (and *inc* in particular) have very high running times and require a lot of memory: Computations on several circuit-based instances as

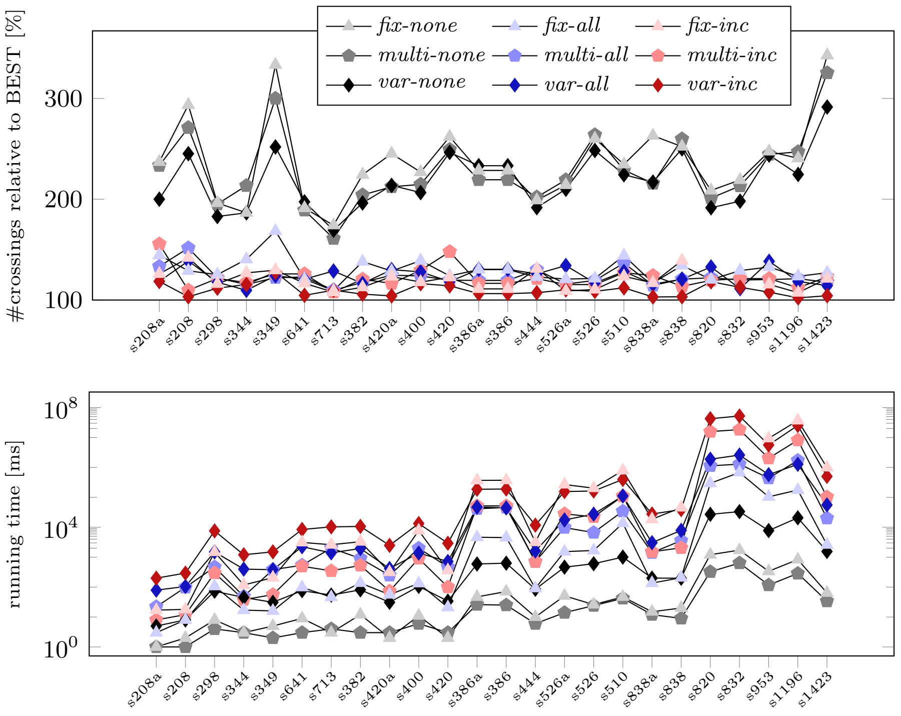


Figure 10. Comparison of the *plm*-variants on the ISCAS-89 instances. The instances s820 and s832 could not be solved with *fix-inc* due to the enforced memory limit.

well as many expanders with $|V(G)| \in \{50, 100\}, \delta = 20$ exceeded the enforced memory limit. For this reason, we did not even attempt to solve the complete (bipartite) instances with *inc*.

Overall, *fix-all* is the fastest *plm*-variant that still benefits from the quality improvements of postprocessing. However, the best compromise between solution quality and speed is provided by the *multi*-variants: Their speed oftentimes matches the speed of the corresponding *fix*-variants. This might be explained by the fact that intermediate planarizations produced by *multi* contain fewer crossings and that there is thus less computational overhead. As our implementation of *multi* performs incremental postprocessing in a fixed embedding setting, *multi-inc* also has a speed advantage over *var-all* and even *multi-all* (which uses postprocessing in a variable embedding setting) on a limited number of sparse instances. Nonetheless, the best results are usually achieved by *var-inc*.

Key takeaways. The postprocessing strategies *all* and *inc* require a lot of time and memory but significantly improve the planarizations produced by the planarization method. The slowest but quality-wise best edge insertion algorithm is *var* whereas *fix* is the fastest but quality-wise worst one. The *multi*-variants serve as a great compromise between them. Overall, the variants *fix-none*, *fix-all*, *multi-all*, *multi-inc*, *var-all*, and *var-inc* (in this order) seem to form sensible choices for different weighings of the two competing measures running time and solution quality.

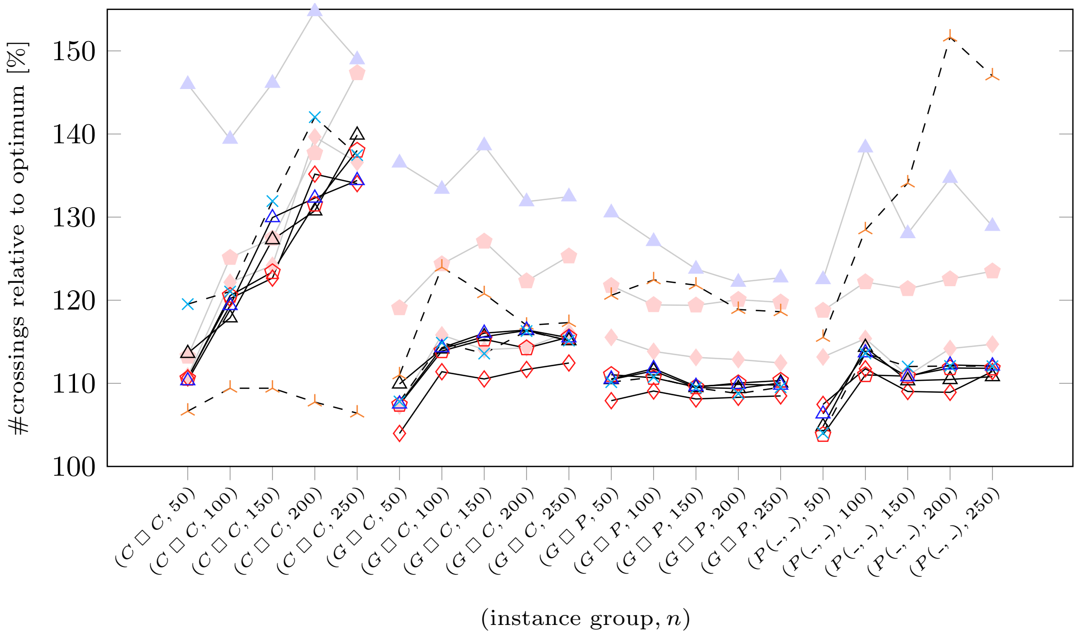
5.3 Improvements via the Star Reinsertion Method

Having discussed all base algorithms, we now focus on *srm*, i.e., the star-insertion-based postprocessing. We first give a general assessment of *srm*'s running time and solution quality. Then we compare specific *plm*-variants using *srm* and conduct an evaluation of *ccm-srm* and *mim-srm*.

We tested *srm* as a postprocessing method for the eight most promising and interesting algorithms that construct an initial planarization, i.e., the fastest base algorithms, the ones with the best solution quality, and those that form a good compromise. More specifically, we consider the three fast algorithms *mim*, *ccm*, and *fix-none*, as well as the more involved *fix-all*, *multi-all*, *multi-inc*, *var-all*, and *var-inc*. In the case of the latter five, a form of postprocessing is already used, and the additional application of *srm* only leads to a small increase in running time, comparatively speaking. In the case of the former three, the additional postprocessing via *srm* significantly increases the running times (*fix-none-srm* becomes even slower than *fix-all-srm*), but the algorithms are still surprisingly fast: On sparse instances, the running times are comparable to *multi-inc* (without *srm*); on dense instances, the algorithms are even faster than *fix-all*. This is especially interesting as all *srm*-enhanced algorithms typically outperform even the best previously known heuristic variant *var-inc* (see Figures 11 and 12). In spite of its simplicity, star insertion in a fixed embedding setting is able to greatly improve intermediate planarizations by inserting multiple edges at once. It provides better results and is faster than edge insertion in a variable embedding setting even if the latter uses incremental postprocessing.

When observing the solution quality of the *srm*-algorithms, the same hierarchy as for the algorithms without *srm* emerges: *fix-none-srm* performs worse than the other *plm*-based *srm*-variants, with *var-inc-srm* providing the best results overall. However, *var-inc-srm* is rarely worth the additional running time since the three significantly faster *mim-srm*, *ccm-srm* and *fix-none-srm* perform similarly well or even surpass it on many instances such as several circuit-based ones and the expanders. In comparison to *mim-srm* for example, *var-inc-srm*'s solution quality difference to BEST is only 1.7% smaller but its median running time is eight times higher (when averaged over all instances).

Ranking the faster *srm*-algorithms by solution quality, however, is difficult as their performance varies greatly on different instances. On Rome, KnownCR, and ISCAS-85 instances as well as small



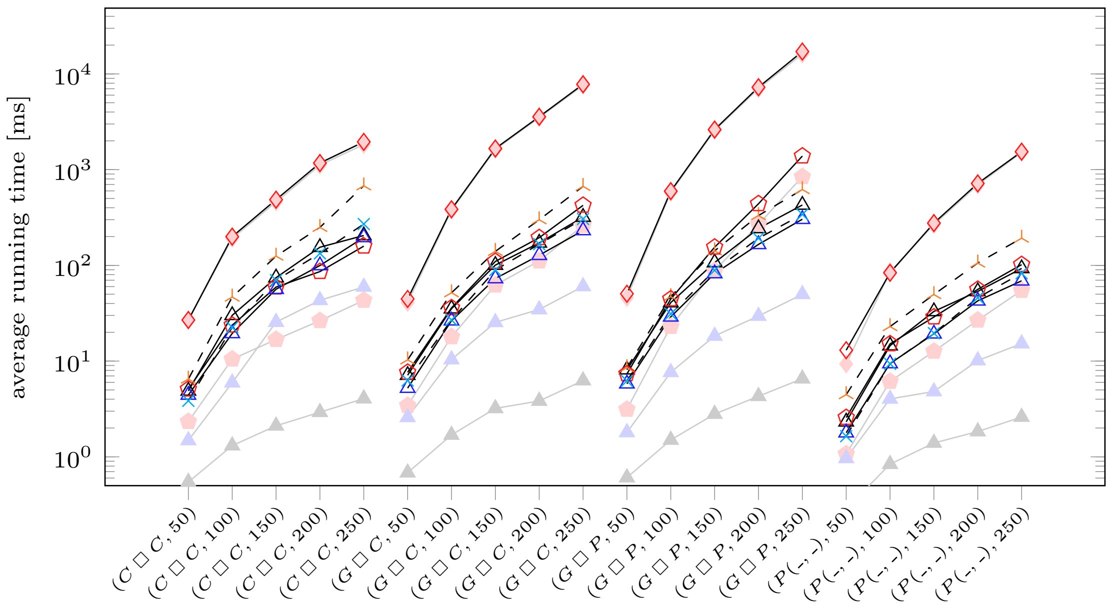


Figure 11. Comparison of the *srm*-variants on the KnownCR instances. The legend of Figure 12 applies. Instances are described on the horizontal axis as tuples of instance type and size n, where n is rounded up to the nearest multiple of fifty. Note that the results of *ccm-srm* heavily depend on the structure of the instance; they also vary a lot across other instance sets (with results of medium quality on average). The plot for *fix-none* (without *srm*) is not depicted in the upper image as its average number of crossings on the KnownCR instances is 218.23% of what BEST achieves.

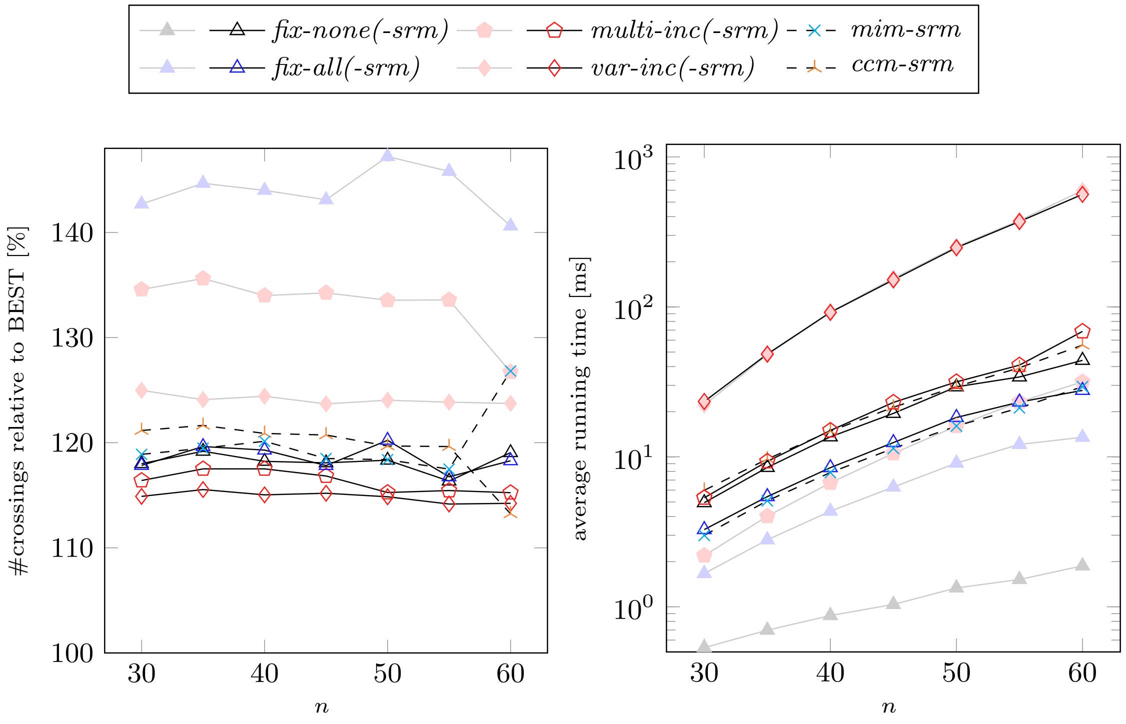


Figure 12. Comparison of the *srm*-variants on the Rome instances. The grayed out plots represent the heuristic variants without *srm*-postprocessing. Instance sizes are rounded up to the nearest multiple of five. The plot for *fix-none* (without *srm*) is not depicted on the left as its average number of crossings on the Rome instances is 202.93% of what BEST achieves. Similarly, we refrain from reprinting *ccm* and *mim* (both without *srm*) here–see Figure 8 instead.

expanders, *mim-srm* and *fix-none-srm* produce better results than *ccm-srm*. On medium-sized expanders and Webcompute instances, it is the other way around. In general, the planarization created by the base algorithm seems to play an important role in determining the quality of the result. This is also indicated by *ccm-srm* on the Petersen graphs: *ccm* without postprocessing performs particularly bad on these instances, and seemingly because of that, *ccm-srm* also produces subpar results—they are even worse than those of *var-inc* without *srm*, which never happens for any other *srm*-algorithm. Nonetheless, *ccm-srm* performs exceptionally well on $C \square C$ due to the high quality improvements by *srm*, emphasizing that the structure of the instances is very relevant to the algorithm's performance (see Figure 11). The running times of the three faster *srm*-algorithms show a clearer picture as they seem to coincide more closely with the quality of the planarization delivered by the base algorithm: While *fix-none-srm* is generally faster than *ccm-srm* on sparse instances, the opposite is true on denser ones. On complete (bipartite) instances, *ccm-srm* becomes even faster than *mim-srm*. However, with a median running time of 10ms across all instances, *mim-srm* is the otherwise fastest among these algorithms (closely followed by *fix-all-srm*, 15ms), and thus we recommend to use it.

Key takeaways. Star-insertion-based postprocessing improves the results of all heuristics. In particular, *mim-srm*, *ccm-srm* and *fix-none-srm* all need less time and produce better results than the best previously known heuristic *var-inc*. Moreover, these algorithms provide solutions that come close to those of *var-inc-srm*—the heuristic with the overall best results—, but they are up to eight times as fast.

5.4 Removal of Non-simple Crossings

To gain insight into the effectiveness of the removal of non-simple crossings, we counted how many of them are detected during each algorithm run. We first present our findings for *plm*, then for *ccm* and *mim*, and finally for the postprocessing routine *srm*. Regardless of the employed heuristic, non-simple crossings occur primarily on dense instances, in particular complete (bipartite) instances.

When inspecting the final planarizations produced by *plm*, it becomes evident that in most cases, the edge-postprocessing strategies *all* and *inc* already remove all non-simple crossings. In fact, on KnownCR instances, not a single such crossing can be found in the final solutions for these algorithm variants. When no edge-postprocessing is used, however, we found and removed such crossings for 11.7–13.2% of all instances (depending on the edge insertion algorithm), with *fix-none* producing more of them than *multi-none* and *var-none*. A maximum of 3410 non-simple crossings were created during the run of *fix-none* on $K_{39,39}$.

For *mim* and *ccm*, we examine the sum of non-simple crossings found and removed after the reinsertion of each star. In comparison to *plm* without postprocessing, these numbers are considerably lower, presumably because inserting multiple edges via star insertion reduces the likelihood of producing such crossings. In particular, when inserting a star as described in Section 3.1, it is not possible to introduce a non-simple crossing between its edges. For *mim*, there are 5.89–9.73% affected instances overall (depending on the variant, 9.51% for *mim-both*), with a maximum of 1916 non-simple crossings for $K_{40,40}$. While *ccm* results in similarly high numbers (6.18% of all instances), the corresponding distribution of non-simple crossings among the instance sets is striking: The algorithm does not produce *any* such crossings on complete (bipartite) instances but showcases a high occurrence rate on circuit-based instances, expanders, and especially KnownCR. The numbers coincide with the performance of *ccm* on the respective instances.

With respect to the *srm*-postprocessing, we only count the non-simple crossings that are removed during the star reinsertion process, with those of the initial planarization being already removed. It can be observed that the totals depend heavily on the quality of the initial planarization—presumably because the creation of a non-simple crossing requires a specific configuration of already existing crossings (cf. Figure 3). Among *srm*-algorithms whose initial planarization is constructed using *all* or *inc*, *fix-all-srm* is the only one with more than 100 instances being affected by non-simple crossings during the star reinsertion phase (121 instances, 1.38%). For the remaining *srm*-algorithms, these numbers are considerably higher: *fix-none-srm* results in the highest amount of affected instances (1094, 12.36%), closely followed by *ccm-srm* (1024, 11.66%), with *mim-srm* producing roughly half as many (536, 6.1%). It is particularly interesting that γ-crossings only occur when running *ccm-srm* on KnownCR instances: There are 30 affected instances, almost all from the $C \square C$ and $G \square C$ sets, that only become affected after running the algorithm several times with random permutations of the inserted stars (see Section 5.5).

Key takeaways. The removal of non-simple crossings can significantly improve the final results of heuristics that do not employ any kind of postprocessing. It can also slightly improve intermediate planarizations during *srm*, potentially speeding up the procedure. However, with more involved postprocessing, non-simple crossings are less likely to occur, and their removal is hence less effective.

5.5 Improvements via Permutations

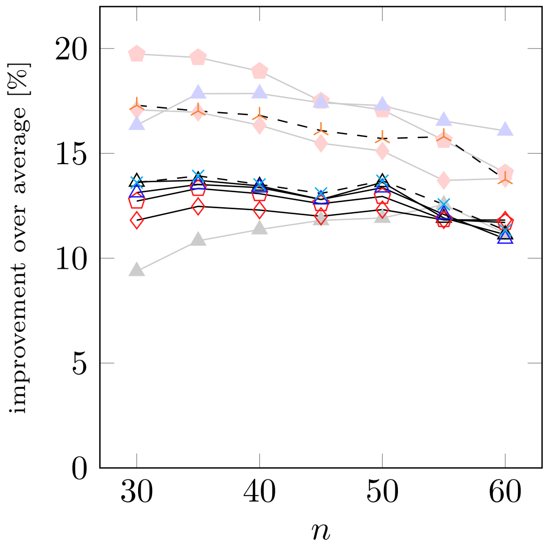
We will consider one last question: Whether multiple runs of the same algorithm with different random permutations of the inserted elements can significantly improve the results. For *plm*, we permute the order in which the deleted edges are inserted, and for *mim*, *ccm* and *srm*, we permute the order of (re)inserted stars. Our experiments compare the effect of 50 random permutations with respect to the Rome, North, Webcompute and KnownCR instance sets. For the larger instances and more time-consuming algorithms, this number of permutations is the limit of what we are able to compute. We focus on the *(relative) improvement* for each instance, calculated as $1 - {}^{\text{best}_{50}}/_{\text{avg}_{50}}$ where best_{50} is the lowest number of crossings and avg_{50} the average number of crossings across 50 permutations. Table 2 lists the average relative improvement for each of the heuristics and instance sets. In the following, we first consider the effect of permutations on heuristics without *srm*, then on heuristics with *srm*, and lastly we rank the best-performing algorithms with and without multiple permutations.

Permutations can significantly improve the results of *mim*, *ccm*, and *plm* without postprocessing while still requiring little time. However, when more time is available, *plm* with postprocessing is clearly preferable. Multiple permutations of *all* and *inc* can be of use if one tries to marginally improve already good solutions. In fact, one can achieve a higher relative improvement when edge-postprocessing is employed: With the exception of very sparse graphs, it is higher for *inc* than for *all*.

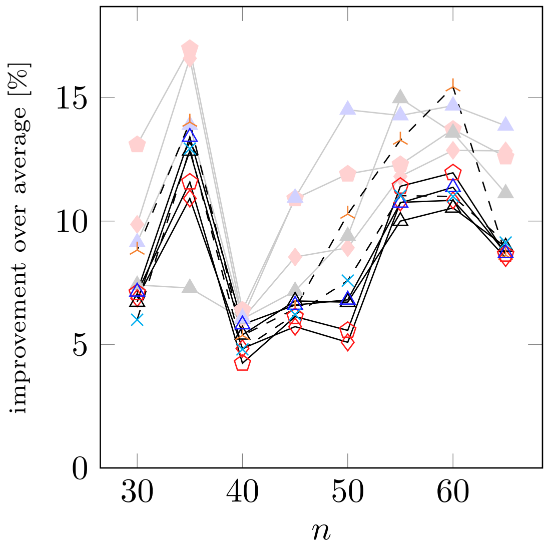
Among the *srm*-algorithms, the relative improvement via permutations is consistently low with little variance; for a comparison with the respective *plm*-variants see Figure 13. The one outlier is

Table 2. Average relative improvement $1 - {}^{\text{best}_{50}}/_{\text{avg}_{50}}$ (in percent) where best_{50} is the lowest number of crossings and avg_{50} the average number of crossings across 50 permutations. Intuitively, this shows the gain in solution quality by 50 permutations compared to a single one.

Algorithm	Rome	North	KnownCR	Webcompute
fix-none	10.60	9.57	4.91	7.60
fix-all	17.20	11.98	7.28	12.39
fix-inc	18.65	12.46	9.99	15.23
multi-none	13.66	11.95	6.14	10.50
multi-all	18.10	12.83	7.87	13.23
multi-inc	19.08	12.98	10.00	14.28
var-none	12.62	11.14	4.95	9.84
var-all	18.44	12.83	8.64	14.08
var-inc	16.57	11.46	8.48	12.43
mim	14.73	12.83	9.41	14.05
ccm	34.13	27.28	27.77	33.16
fix-none-srm	13.49	8.80	5.32	8.90
fix-all-srm	13.23	9.25	5.02	8.13
multi-all-srm	12.84	8.31	5.29	7.52
multi-inc-srm	12.93	8.93	6.72	6.90
var-all-srm	13.08	8.94	5.31	7.69
var-inc-srm	12.12	8.45	6.50	6.45
mim-srm	13.59	8.77	5.63	7.81
ccm-srm	16.88	11.03	15.19	10.38



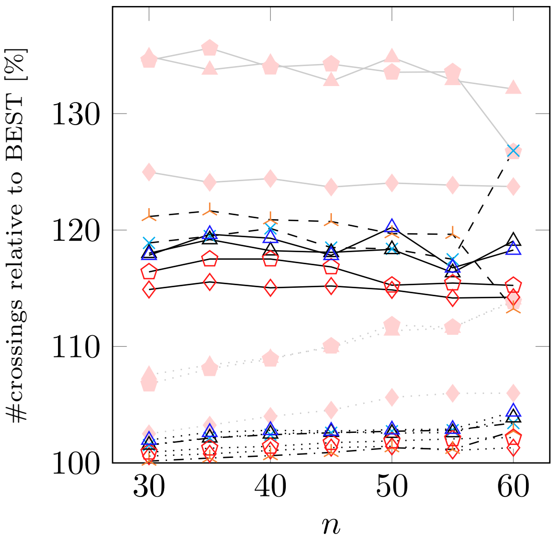
(a) Rome



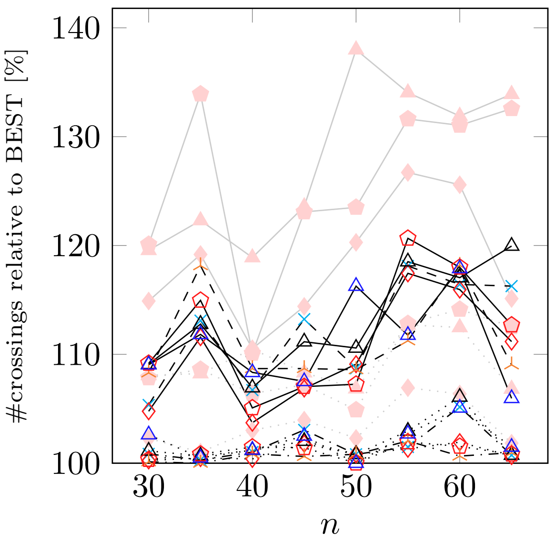
(b) North

Figure 13. Comparison of relative improvements for 50 permutations over their average on the Rome and North instances. The legend of Figure 12 applies.

—— *average*	– × – *mim-srm*	△ *fix-none-srm*	⬠ *multi-inc-srm*	▲ *fix-inc*	♦ *var-inc*
······ *best-of-50*	– ⅄ – *ccm-srm*	△ *fix-all-srm*	◇ *var-inc-srm*	⬟ *multi-inc*	



(a) Rome



(b) North

Figure 14. Permutation-based comparison of high-solution-quality heuristics on the Rome and North instances. Solid (and dashed) lines denote the average number of crossings among 50 permutations; dotted (and dash-dotted) lines denote the lowest number of crossings after 50 permutations.

ccm-srm, which achieves the greatest relative improvements for 50 permutations. Its average value on Rome graphs, for example, is 16.88% while other algorithms never leave the range of 5–14%. This coincides with the fact that the solutions produced by *ccm* (without *srm*) can already be greatly improved by multiple permutations: Only 10 permutations already do so by around 20%, 50 permutations by around 30%. Note, however, that we initialize all permutations of *ccm-srm* with a fixed small chordless cycle instead of a fixed maximal planar subgraph. This allows for greater variance in the solutions of *ccm-srm* and makes it difficult to compare the results to other *srm*-algorithms.

The general trend of high-solution-quality algorithms, taking multiple permutations into account, is shown in Figure 14: A single permutation of *mim-srm* or *ccm-srm* will yield better solutions than a *plm*-variant with incremental postprocessing (but no *srm*). Two layers of postprocessing, i.e., *-all-srm* or *-inc-srm*, improve the results even more. Solutions resulting from 50 permutations are in a tier of their own, with *srm*-heuristics achieving higher quality than those without. Overall, 50 permutations of *mim-srm* or *ccm-srm* provide some of the best results while taking a lot less time than other algorithms in their category. Consider, e.g., the Rome instances in a 50-permutations setting; *var-inc-srm* can reduce the average solution quality difference to BEST by only 1.2% more than *mim-srm*, but its median running time is ten times as high.

Key takeaways. For *mim*, *ccm*, and *plm* without postprocessing, it is more fruitful to employ postprocessing rather than multiple permutations. While permutations improve results of all heuristics (see Table 2 for the percentages), increasing their amount has diminishing returns. When comparing heuristics with 50 permutations, *var-inc-srm* delivers the overall best results, however, *mim-srm* as well as *ccm-srm* provide similar results with much lower running times.

6 Conclusion

Our in-depth experimental evaluation not only corroborates the results of previous papers [10, 17] but also provides new insights into the performance of star insertion in crossing minimization heuristics. We presented the novel heuristic *mim*, which proceeds similarly to the planarization method but inserts most edges by reinserting one of their endpoints as a star. Whenever neither endpoint is a cut vertex of the initial planar subgraph, the endpoint can be chosen freely, and our experiments indicate that reinserting *both* endpoints one after another provides the best results. In general, *mim* performs better than the basic heuristics from [10, 17] that have a similarly low running time (i.e., *ccm* and *fix-none*).

A central observation is that postprocessing via star insertion (*srm*) can greatly improve the planarizations resulting from fast heuristics: *mim-srm*, *ccm-srm*, and *fix-none-srm* are all faster than the previously best-performing heuristic *var-inc* and provide better results. By inserting multiple adjacent edges at once, star (re-)insertion changes the planarization and its underlying graph decomposition in a way that is sufficient to properly explore the search space and find good solutions. Fixed embedding star insertion is thus preferable over the much slower insertion of edges (or even stars) in a variable embedding setting.

We note that many heuristics—in particular those without edge-wise postprocessing—are prone to create non-simple crossings. Such crossings can be detected and it is worthwhile to remove them in order to speed up the procedure and improve the results. Lastly, multiple permutations are beneficial for heuristics that already employ postprocessing. In particular, their application to *mim-srm* and *ccm-srm* provides very high solution quality at moderate running times.

References

[1] C. Batini, M. Talamo, and R. Tamassia. Computer aided layout of entity relationship diagrams. *J. Syst. Softw.*, 4(2-3):163–173, 1984. doi:10.1016/0164-1212(84)90006-2.

[2] G. D. Battista, A. Garg, G. Liotta, A. Parise, R. Tamassia, E. Tassinari, F. Vargiu, and L. Vismara. Drawing directed acyclic graphs: An experimental study. *Int. J. Comput. Geom. Appl.*, 10(6):623–648, 2000. doi:10.1142/S0218195900000358.

[3] G. D. Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom.*, 7:303–325, 1997. doi:10.1016/S0925-7721(96)00005-3.

[4] F. Brglez, D. Bryan, and K. Kozminski. Notes on the ISCAS'89 benchmark circuits. Technical report, North-Carolina State University, Oct 1989.

[5] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational circuits and a targeted translator in FORTRAN. In *Proc. ISCAS; Special Session on ATPG and Fault Simulation*, pages 151–158, Jun 1985.

[6] C. Buchheim, M. Chimani, D. Ebner, C. Gutwenger, M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. A branch-and-cut approach to the crossing number problem. *Discret. Optim.*, 5(2):373–388, 2008. doi:10.1016/j.disopt.2007.05.006.

[7] S. Cabello and B. Mohar. Crossing number and weighted crossing number of near-planar graphs. *Algorithmica*, 60(3):484–504, 2011. doi:10.1007/s00453-009-9357-5.

[8] P. Chalermsook and A. Schmid. Finding triangles for maximum planar subgraphs. In *Proc. WALCOM 2017*, volume 10167 of *LNCS*, pages 373–384. Springer, 2017. doi:10.1007/978-3-319-53925-6_29.

[9] M. Chimani and C. Gutwenger. Non-planar core reduction of graphs. *Discret. Math.*, 309(7):1838–1855, 2009. doi:10.1016/j.disc.2007.12.078.

[10] M. Chimani and C. Gutwenger. Advances in the planarization method: Effective multiple edge insertions. *J. Graph Algorithms Appl.*, 16(3):729–757, 2012. doi:10.7155/jgaa.00264.

[11] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. The Open Graph Drawing Framework (OGDF). In *Handbook on Graph Drawing and Visualization*, pages 543–569. Chapman and Hall/CRC, 2013. doi:10.1201/b15385.

[12] M. Chimani, C. Gutwenger, and P. Mutzel. Experiments on exact crossing minimization using column generation. *ACM J. Exp. Algorithmics*, 14, 2009. doi:10.1145/1498698.1564504.

[13] M. Chimani, C. Gutwenger, P. Mutzel, and C. Wolf. Inserting a vertex into a planar graph. In *Proc. SODA 2009*, pages 375–383. SIAM, 2009. doi:10.1137/1.9781611973068.42.

[14] M. Chimani and P. Hlinený. A tighter insertion-based approximation of the crossing number. *J. Comb. Optim.*, 33(4):1183–1225, 2017. doi:10.1007/s10878-016-0030-z.

[15] M. Chimani, P. Mutzel, and I. M. Bomze. A new approach to exact crossing minimization. In *Proc. ESA 2008*, volume 5193 of *LNCS*, pages 284–296. Springer, 2008. doi:10.1007/978-3-540-87744-8_24.

[16] M. Chimani and T. Wiedera. An ILP-based proof system for the crossing number problem. In *Proc. ESA 2016*, volume 57 of *LIPIcs*, pages 29:1–29:13, 2016. doi:10.4230/LIPIcs.ESA.2016.29.

[17] K. Clancy, M. Haythorpe, and A. Newcombe. An effective crossing minimisation heuristic based on star insertion. *J. Graph Algorithms Appl.*, 23(2):135–166, 2019. doi:10.7155/jgaa.00487.

[18] F. Corno, M. S. Reorda, and G. Squillero. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Des. Test Comput.*, 17(3):44–53, 2000. doi:10.1109/54.867894.

[19] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983. doi:10.1137/0604033.

[20] C. Gutwenger. *Application of SPQR-Trees in the Planarization Approach for Drawing Graphs*. PhD thesis, TU Dortmund, Dortmund, Germany, 2010. URL: http://hdl.handle.net/2003/27430.

[21] C. Gutwenger and P. Mutzel. An experimental study of crossing minimization heuristics. In *GD 2003: Revised Papers*, volume 2912 of *LNCS*, pages 13–24. Springer, 2003. doi:10.1007/978-3-540-24595-7_2.

[22] C. Gutwenger, P. Mutzel, and R. Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005. doi:10.1007/s00453-004-1128-8.

[23] P. Hlinený. Crossing number is hard for cubic graphs. *J. Comb. Theory, Ser. B*, 96(4):455–471, 2006. doi:10.1016/j.jctb.2005.09.009.

[24] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973. doi:10.1145/362248.362272.

[25] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

[26] M. Schaefer. The graph crossing number and its variants: A survey. *Electronic Journal of Combinatorics*, DS21 version 6:83–84, 2021. doi:10.37236/2713.

[27] A. Steger and N. C. Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computing*, 8(4):377–396, 1999. doi:10.1017/S0963548399003867.

[28] T. Ziegler. *Crossing minimization in automatic graph drawing*. PhD thesis, Saarland University, Saarbrücken, Germany, 2001. URL: http://d-nb.info/961610808.