Advances in the Planarization Method: Effective Multiple Edge Insertions

Markus Chimani[1,⋆] and Carsten Gutwenger[2]

[1] Inst. of Computer Science, FSU Jena
markus.chimani@uni-jena.de
[2] Dep. of Computer Science, TU Dortmund
carsten.gutwenger@tu-dortmund.de

Abstract. The planarization method is the strongest known method to heuristically find good solutions to the general crossing number problem in graphs: starting from a planar subgraph, one iteratively inserts edges, representing crossings via dummy nodes. In the recent years, several improvements both from the practical and the theoretical point of view have been made. We review these advances and conduct an extensive study of the algorithms' practical implications. Thereby, we present the first implementation of an approximation algorithm for the crossing number problem of general graphs, and compare the obtained results with known exact crossing number solutions.

1 Introduction

Given a graph $G = (V, E)$, the *crossing number* problem asks how to draw G into the plane with the fewest possible number of edge-crossings. The *planarization method* is the probably best known and most successful heuristic to tackle the crossing number problem in practice. In its simplest form it runs in two phases: first, a (large) planar subgraph $G' = (V, E') \subseteq G$ is computed. Then, the temporarily removed edges $F := E \setminus E'$ are re-inserted one after another, each time solving a *single edge insertion* problem. This problem can be stated as follows: Let H be a planar graph, and e an edge not yet in H. We search for a smallest planar graph H^+ which represents a drawing of $H + e$ where edge crossings are replaced by dummy nodes of degree 4, and all these crossings occur on the edge e. Hence, when removing the image of e from H^+, we obtain a planar embedded H. Using this method, each edge of F is inserted in a planar graph until we obtain a *planarization* G^+, representing G in a planar way by using dummy nodes for crossings.

In the first proposal [1] of this heuristic, the insertion problem was considered w.r.t. a *fixed* embedding (cyclic order of the edges around their incident nodes) of the planar graph H. (I.e., after obtaining the planar subgraph G', one embedding of G' is fixed and retained throughout the whole insertion phase.) A simple linear-time BFS-algorithm in the dual graph of H suffices to find an optimal solution.

⋆ Markus Chimani was funded by a Carl-Zeiss-Foundation juniorprofessorship.

M. van Kreveld and B. Speckmann (Eds.): GD 2011, LNCS 7034, pp. 87–98, 2012.

Later, and rather surprisingly, it was shown in [14] that there exists a linear-time algorithm, using the SPQR-tree datastructure, which finds the optimal insertion path for e over all possible planar embeddings of H. In [13] it was shown that this approach is in practice vastly superior to the former in terms of the overall obtained number of crossings.

In recent years it was furthermore shown that there exists a (rather complex) insertion algorithm to optimally insert a vertex with all its incident edges into a planar graph [4] (*vertex insertion*), while it is NP-hard to insert an arbitrary set of edges simultaneously [18] (*multiple edge insertion*).

Most interestingly, the single edge insertion problem (over all possible embeddings of H) is known to approximate the crossing number of $H+e$ within a factor of $\Delta/2$ (where Δ is the graph's maximum degree) [15, 2], and also the vertex insertion problem approximates the crossing number of the resulting graph [6]. In particular, the proof of the latter can be generalized to show that an optimal multiple edge insertion solution—w.r.t. an edge set F—would approximate the crossing number of $G' + F$ within a factor only dependent on Δ and $|F|$ [6].

Hence, the question arose whether this multiple edge insertion problem can be efficiently approximated. After a rather complicated approach in [8], a simpler and at the same time approximation-wise stronger algorithm was presented only recently [5]. The algorithm reuses concepts of the SPQR-tree based single edge insertion and seems simple enough to be implemented and used in practice. The latter paper also shows that the traditional iterative single edge insertion algorithm cannot be an approximation strategy for the crossing number of G.

Contribution. In this paper we present recent advances of the planarization approach from a practical point of view. On the one hand, we show how to improve on the traditional approach of iteratively inserting single edges, via the use of strong postprocessing routines. On the other hand, we give the first practical implementation of a simultaneous multiple edge insertion algorithm—hence, this is also the first practical study of any crossing number approximation algorithm for arbitrary graphs. By considering graph classes of known crossing numbers (either from theory or from the application of the currently strongest branch-and-cut based exact crossing minimization algorithm [7]) we can deduce a practically very good performance of these heuristics, as they usually find optimum, or at least very-close-to-optimum, solutions.

2 Planarization Approach

In order to present our algorithmic choices and modifications, we first have to briefly introduce two central decomposition structures, used in all algorithms dealing with the insertion problem over all possible embeddings of H. In the above sketched planarization scheme, we can assume that the original graph G is connected—otherwise the crossing number problem decomposes into multiple independent problems. Furthermore the initial planar subgraph G' can be assumed to be maximal and hence also connected. For the single edge insertion algorithms, we will usually consider any intermediate graph H; for the multiple edge insertion algorithm we set $H := G'$.

First, we use the well known *BC-tree* $\mathcal{B} = \mathcal{B}(H)$ of H which is a tree with two different node types B and C: For each cut vertex (maximal two-connected subgraph or bridge, summarized under the term *block*) in H, $\mathcal{B}$ contains a unique corresponding C-node (B-node, respectively). Two nodes in $\mathcal{B}$ are adjacent if and only if they correspond to a block and a cut vertex, where the former contains the latter. We can construct such a linear-sized BC-tree $\mathcal{B}$ in linear time.

Based thereon, we can further decompose non-trivial blocks (i.e., non-bridges) via SPQR-trees [10]: While they are more complicated than BC-trees, they also only require linear size and can be constructed in linear time [16,12]. This datastructure is particularly interesting, as it directly encodes all (exponentially many) planar embeddings of its underlying block. We use the definition from [3,5] which does not use Q-nodes, and therefore call the decomposition tree $\mathcal{T} = \mathcal{T}(H')$ of a non-trivial block H' *SPR-tree* for conciseness. Chiefly summarizing, each tree node corresponds to a *skeleton*, which is a "sketch" of H' where certain subgraphs are replaced by virtual edges. By repeatedly merging the skeletons of adjacent nodes (at their virtual edges representing each other), we can obtain the original graph, and each virtual edge hence represents a 2-cut (*split pair*) in H'. Most importantly, a skeleton can only be one of three types: The skeleton of an S-node ("serial") is a simple cycle; the skeleton of a P-node ("parallel") consists of two vertices and multiple edges between them; the skeleton of an R-node is a simple triconnected graph. Note that a planar triconnected graph has a unique embedding (up to mirroring).

In the algorithmic description of the multiple edge insertion approximation algorithm [5], an amalgamated version of these trees, the so-called *con-tree*, is considered: a BC-tree, directly storing SPR-trees at the non-trivial B-nodes.

Single Edge Insertion. We will briefly recapitulate the central ingredients of the exact linear-time algorithm by Gutwenger et al. [14] to solve the single edge insertion problem over all possible embeddings of H. Let v_1, v_2 be the vertices we want to connect in H via a new edge. First consider a fixed embedding of H and let H_D be its dual. We define an *insertion path* to be a path in H_D connecting a face incident to v_1 with a face incident to v_2. The length of this path is then the number of edge crossings necessary to insert the edge $\{v_1, v_2\}$ into embedded H along this path; each dual edge in the insertion path corresponds to an edge in H that is to be crossed. We can directly compute the shortest insertion path via standard breadth-first search (BFS).

Now consider H with variable embedding. Let L be the unique shortest path in $\mathcal{B}(H)$ from a B-node containing v_1 to a B-node containing v_2. The optimal insertion path for $\{v_1, v_2\}$ in G can be obtained by concatenating the optimal insertion paths within the (non-trivial) blocks on this path L; we can always nest blocks at a common cut vertex into each other such that there arise no additional crossings. For a block H' represented by a B-node on L, let $v_i^{H'}$, $i = 1, 2$, denote v_i if $v_i \in V(H')$, or the cut vertex in H' closest to v_i otherwise. It remains to, for each non-trivial block H', find optimal insertion paths from any face incident to $v_1^{H'}$ to any face incident to $v_2^{H'}$.

Therefore, let $Q_{H'}$ be the unique shortest path in $\mathcal{T}(H')$ from a skeleton containing $v_1^{H'}$ to a skeleton containing $v_2^{H'}$. It was shown in [14] that only the embeddings of the skeletons along $Q_{H'}$ matter. In a nutshell, the algorithm walks along these skeletons and fixes suitable embeddings for the skeletons, one after another. Finally, an optimal embedding is found and fixed, and one can use the simple BFS algorithm on the dual graph to insert the edge $\{v_1^{H'}, v_2^{H'}\}$ optimally.

In the following, we can consider a *con-chain* Q of the edge $\{v_1, v_2\}$ as an extended version of L, where the "subpaths" $Q_{H'}$ are stored at each non-trivial block H' along L.

Multiple Edge Insertion. Let us briefly review the approximation algorithm for the multiple edge insertion problem by Chimani and Hliněný [5]. Let $H := G'$ be the initial planar subgraph of G into which to insert the edges $F = \{e_i\}_{1 \le i \le |F|}$. Assume we could independently insert each edge $e_i \in F$ into H. Using the above algorithm for single edge insertions, we would obtain a con-chain Q_i for each edge e_i, and therefore a so-called *embedding preference* for each node on Q_i w.r.t. e_i. Coarsely speaking, we obtain a common embedding of H via a voting scheme on the (possibly conflicting) embedding preferences per con-tree node, ensuring that at any node at least one preference is satisfied. After realizing the so-chosen embedding, we can once again use the simple BFS algorithm in the dual graph to insert the edges into this fixed embedding.

The prove-wise crucial part in the algorithm is that any two con-chains Q_i, Q_j are either disjoint or they intersect in one sub-chain. Hence, two con-chains (think of simple paths) "deviate" at at most two nodes in the con-tree (think of a regular tree): once when the two paths come together and once when they part. Roughly speaking, it is shown in [5] that the embedding preferences for the tree nodes can differ only at these two "places" (called *passes*), whereby the exact definition of *pass* is quite involved and might in fact span over up to three con-tree nodes. Yet, overall we can bound the number of nodes where some con-chains disagree on the embedding preference, as well as the additionally necessary number of crossings to route an edge through a skeleton that is differently embedded than desired. This gives an approximation factor for the optimal multiple edge insertion w.r.t. G' and F, and, subsequently, for the crossing number of G.

It remains to clarify what an *embedding preference* actually is: Observe that S-nodes do not allow different embeddings of their skeletons. For an R-node (a triconnected planar graph), we have only a unique planar embedding and its mirror. For a P-node, each inserted edge may want two particular edges of the skeleton to be cyclicly adjacent (in, say, clockwise direction). Finally, for a C-node each inserted edge may want a particular incident face in an adjacent block to be identified with a particular incident face in another adjacent block.

3 Engineering

Iterative Single Edge Insertion and Postprocessing. In the traditional planarization heuristic, we will "simply" insert the temporarily removed edges

F one after another into the planar subgraph. After each insertion, we replace the arising crossings by dummy nodes, and hence proceed with a planar graph. There are various ways to fine-tune the obtained result via postprocessing, as already discussed in [13]. The simplest—and in fact quite effective—variant is to start the insertion process multiple times, each time with a different, randomized order of F. Additionally, each such insertion run can be improved: After having inserted all edges, we can again remove some original edge e from the planarization (i.e., we remove all the subedges and dummynodes that represent e), and re-insert it, possibly requiring fewer crossings. For this operation, we can consider either the inserted edges F (*ins*), all edges (*all*), or the x% of the edges with the most crossings (*most*, for some constant x). In [13] it was shown, that these approaches lead to greatly improved results.

Herein, we propose a further improvement on these methods. The *incremental (inc)* strategy basically applies the *all* strategy after each single insertion step. I.e., after the insertion of an edge $e \in F$, we try to remove and reinsert every other edge already in the graph, in order to obtain a better crossing number, before proceeding with the next edge from F. We will see, that this approach again dominates the previously best strategy *all*, though at the cost of a vastly increased running time.

Note that all these strategies—when applied in a fixed embedding setting—are also applicable to the multi-edge insertion problem, after fixing an embedding into which all edges F need to be inserted. Formally, the *inc* setting has to restrict itself to only try to reinsert the edges F, in order to retain the approximation guarantee. Interestingly, after having obtained a postprocessed solution in the fixed embedding, we can run the *all* postprocessing where the graph's embedding may change, i.e., using the optimal edge insertion over all possible embeddings! As the solution value never decreases, the algorithm retains its approximation guarantee and improves the number of crossings in practice.

From the approximation point of view, we can observe that the first part of the algorithm (fixing a suitable overall embedding) tries to minimize the number of crossings between F and G', while the postprocessing routines most importantly try to reduce the number of crossings between edges of F—their quantity can only be estimated as $\binom{|F|}{2}$ in the formal quality guarantee.

Implementing Multiple Edge Insertion. In [5], certain aspects of the multiple edge insertion algorithm are described to be suitable for a comparatively smooth approximation proof. When implementing the algorithm, we take some different, though completely equivalent, routes. A main point of deviation is the consideration of *dirty passes*, i.e., con-tree node tuples where multiple insertion paths disagree on their preferred embedding. We highlight the two main divergent choices here. Overall, our viewpoint allows a quite simpler implementation than would be easily deduced from the theoretical proofs of [5] alone.

Con-Tree. Originally, an amalgamated version of BC- and SPR-trees is proposed, which allows to talk about a single chain (path) for each inserted edge. In the implementation, we perform the algorithm differently: First, we compute a

suitable combinatorial embedding for each non-trivial block independently. Only then, we consider the C-nodes at which the blocks are joined. From the formal definition of dirty passes, we can easily deduce that C-nodes do not interact with other nodes in terms of realized embedding preferences, and hence we can independently choose which faces to embed into each other at cut vertices, after fixing the embeddings of the incident blocks.

This modification allows us to consider only two-connected graphs and SPR-trees in the following, vastly simplifying implementation details as most of the infrastructure necessary for single edge insertions can be reused.

Merging the Embedding & Repairing Dirty Passes. In [5], the formal definition of dirty passes needs to group nodes as tuples of 1–3 SPR-tree nodes and requires a tie-breaking to prohibit invalid node tuple overlaps. Yet, from the proofs it becomes clear that this is merely necessary to correctly estimate the *number* of these passes. Within the algorithm, these passes are only detected in order to identify possible flips to prohibit too many such situations. For this purpose alone, a much simpler strategy suffices: Usually, we consider one insertion path after another: We traverse its SPR-nodes and fix the embedding of each skeleton along this path as preferred. When a visited node already has a fixed embedding, we (coarsely speaking) try to flip the predecessor nodes of our current path in order to avoid dirty passes. Instead of checking the full case distinction in the dirty pass definition, it suffices to consider the case where the currently visited nodes ν and its predecessor (disregarding S-nodes) μ are P- and/or R-nodes:

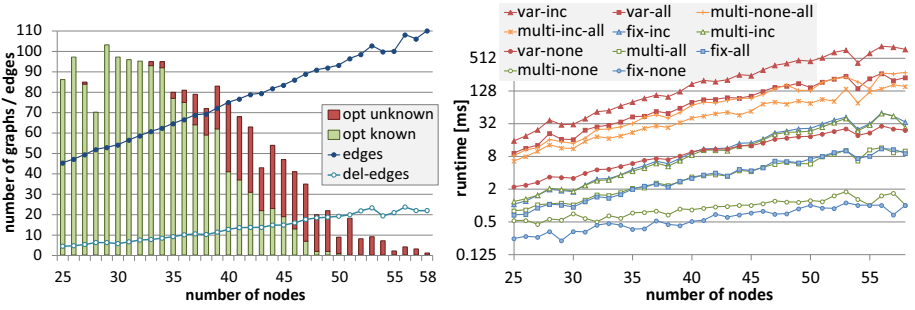
We say an embedding preference at a P-node *agrees* with a fixed embedding of this node's skeleton, if the specified two edges occur clockwise neighboringly. An embedding preference of an R-node is simply a binary flag specifying whether to use a "default" planar embedding of the node's skeleton or the "mirror" (only these two embeddings exist). Now, we only have to flip μ and its predecessors along the insertion path iff μ and ν are *switching*, i.e., the new embedding preferences agree with the already fixed embedding of one of these two nodes, and agrees with the *flipped* embedding of the other node.

Doing this for all such pairs ν, μ then also repairs dirty passes on node triples, if at all possible. In all other cases of dirty passes, no flip can improve the situation anyway and hence is not necessary. It is understood that this procedure performs the same flips as the more abstract merge routine described in [5], and hence the implementation retains the approximation guarantee.

4 Experiments

Experimental Setup. We implemented all algorithms using the C++ library OGDF[1] and ran our experiments on a Linux system with an Intel Core i7 (2.67 GHz) processor and 12 GB RAM. For each instance, all edge insertion algorithms were called with the same, pre-computed maximal planar subgraph, computed via the PQ-tree based planar subgraph algorithm [17] (best of 250

[1] Open Graph Drawing Framework, see `http://www.ogdf.net`



(a) statistics (b) runtimes

Fig. 1. *Rome* graphs

random runs, i.e., random choices of the initial *st*-edge for the numbering) and iteratively adding removed edges afterwards if they do not destroy planarity.

We consider four benchmark sets, the first two of which are the well-known *Rome* library [9] and *AT&T* graphs (available at `http://graphdrawing.org/data.html`). We first applied a reduction strategy that removes parallel edges, self-loops, and planar biconnected components, and reduces paths in the graph to single edges (unless this introduces parallel edges). We consider all remaining non-planar connected components with at least 25 nodes and at least two edges removed in the computed planar subgraph, which are 1843 graphs in the *Rome* set (25–58 nodes) and 311 graphs in the *AT&T* set (25–312 nodes). The *ISCA* graphs are hypergraphs taken from the ISCA'85 benchmark set of real world electrical networks, transformed into traditional graphs by substituting each hyperedge h by a new hypernode connected to all nodes contained in h, connecting all inputs (outputs) to a new node s_{in} (s_{out}, resp.), and introducing the edge $(s_{\text{in}}, s_{\text{out}})$. We used the same reduction and selection as described above leading to 20 graphs (25–223 nodes). Finally, the *KnownCR* graphs [11] are a collection of 1946 graphs with known crossing numbers (by proofs), consisting of generalized Petersen graphs ($P(m,2)$, $P(m,3)$) and products of cycles C_n, paths P_n, and 5-vertex graphs G_i ($C_m \times C_n$, $G_i \times P_n$, $G_i \times C_n$); these graphs have between 9 and 250 nodes. Our whole benchmark set can be downloaded from `http://ls11-www.cs.uni-dortmund.de/people/gutweng/planexp.zip`.

Rome Graphs. Fig. 1(a) gives an overview on the Rome benchmark set, displaying the number of graphs and average number of edges per node count. Furthermore, it shows the average number of edges deleted in the planar subgraphs and for how many of the graphs we know the exact crossing number from the branch-and-cut algorithm presented in [7, 3].

We first consider the effect of postprocessing; see Fig. 2. We compare the results with the *best* known results (from our experiments and the branch-and-cut algorithm [3]) and show the relative difference between heuristic and best solution. Since we know the exact solutions for many of the graphs, this gives a very good impression on the actual quality of the heuristics. We note that the

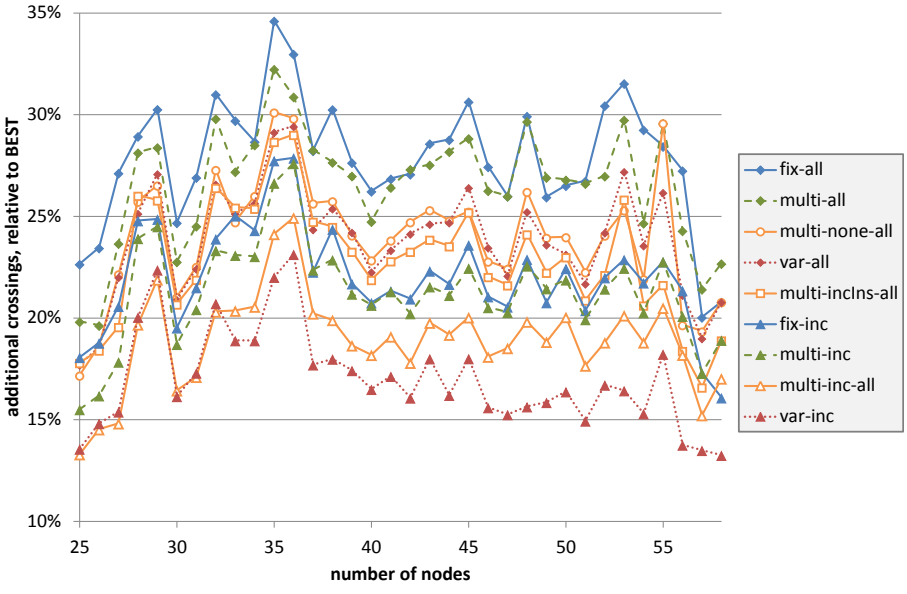


Fig. 2. Number of crossings for *Rome* graphs, relative to BEST known solutions

exact algorithm is clearly slower than any of the considered heuristics by orders of magnitudes, cf. [7,3]. As already observed in [13], postprocessing helps a lot, and this also holds for multiple edge insertion (the values without postprocessing lie between 60–70%). Our new incremental postprocessing achieves clearly better results than the previously best *all*, for all edge insertion strategies. We also observed that the advantage of *multi* over *fix* is large without postprocessing, but becomes smaller and smaller the more postprocessing is applied, since the postprocessing becomes the dominating factor and is the same for both.

Inspired by this observation, we experimented with an additional postprocessing for the *multi* strategy, where we reused the postprocessing with variable embedding; see Fig. 2. The variants *multi-none-all* and *multi-inc-all* perform *multi* with no or incremental postprocessing plus postprocessing with variable embedding afterwards; *multi-incIns-all* restricts the incremental postprocessing to the inserted edges. Whereas *multi-none-all* and *multi-incIns-all*—which retain the approximation guarantee—are about as good as *var-all*, *multi-inc-all*—which in theory does not give those guarantees—comes close to *var-inc* (for larger graphs, it lies between *var-all* and *var-inc*).

In practice, we want to obtain good solutions quickly, hence it is important to look at the runtimes; see Fig. 1(b). We can see that the overhead of *multi* compared to *fix* is small, and even becomes negligible if postprocessing is used. The *var* variants are always clearly slower, as they require a new SPR-decomposition after each edge insertion, whereas *multi* uses only a single such decomposition. The *inc* variants take about 2–4 times longer than *all*, which is acceptable regarding the achieved improvements in quality. For our postprocessing variants

Fig. 3. Effect of permutations on number of crossings (*Rome* graphs)

for *multi* with additional *var*-postprocessing, we observe that more intensive postprocessing with fixed embedding reduces the effort required with the time-consuming *var*-postprocessing and results in smaller runtimes (i.e., *multi-inc-all* is faster than *multi-none-all*). Since *multi-inc* requires a similar runtime as *var-none* but is in quality even better than *var-all* (the previously quality-wise best known heuristic variant), it is a very good choice in practice.

Fig. 3 finally studies the effect of randomly permuting the edges to be inserted (we considered 100 permutations here). The diagram shows the relative reduction of the gap between single run and best solution (hence, 100% means that 100 permutations led to the best solution). The main message is that permutations without postprocessing are not very effective, whereas the combination of postprocessing and permutations always gets significant improvements. The incremental postprocessing variant does not only lead to best results, but is also the most effective one in combination with permutations.

KnownCR Graphs. This collection allows us to further compare the heuristic results with actual crossing numbers. Fig. 4 summarizes our findings for some selected heuristics, showing the average relative deviation from the crossing number for the different graph classes. The class $P(m, 2)$ (all whose graphs have crossing number 2 or 3) could be solved to optimality by all heuristics and we omit it in the diagram. For the classes $P(m, 3)$, $G_i \times C_n$, and $G_i \times P_n$, all heuristics perform well, being only 2-13% away from the optimum, and their order with respect to quality is as expected. The class $C_n \times C_m$ shows some unusual behavior: Without permutations, the insertion strategy seems to have only a very small influence on the solution quality; surprisingly, with 100 permutations, *fix*

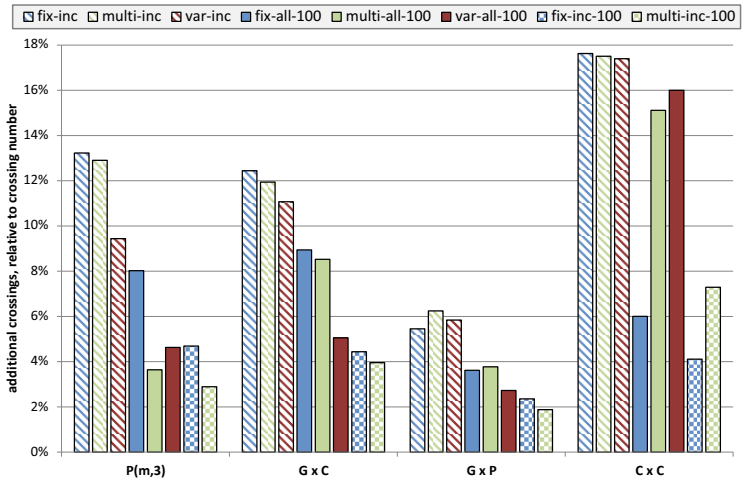


Fig. 4. Number of crossings for *KnownCR* graphs, relative to crossing number

is superior to both *multi* and *var*. Analyzing the data, we see that this happens only for a few graphs: the distinct runs of *fix* usually find slightly worse solutions than *multi* or *var*, but in some rare cases a much better solution is found. We assume that this is caused by the fact that accepting worse intermediate solutions while inserting the edges can lead to a better final solution.

AT&T Graphs. Whereas the *Rome* graphs are fairly homogeneous graphs with a simple structure and *KnownCR* consists of artificial graphs, the *AT&T* graphs are real-world graphs with quite diverse structures. For analyzing the results, we group the graphs according to the best found solutions (the first group contains graphs with 0,...,24 crossings; the last group with 700,...,799 crossings). Fig. 5 shows the relative difference between heuristic and best solution. We can confirm that incremental postprocessing clearly dominates *all* for all edge insertion strategies, and *var-inc* is by far the best strategy both without and with (*var-inc-100*) permutations. Multiple edge insertion is also slightly better than *fix*.

However, the domination of *var* comes at a price: Whereas *fix* and *multi* take about the same runtime, *var* is more than 10 times slower. Hence, *multi-inc* is again a good compromise, as it is even clearly faster (3–10 times) than *var-all*.

ISCA Graphs. We focus on the *multi* and *var* methods. Fig. 6 shows the relative difference between heuristic and best solution, for each graph in the benchmark set separately. The graphs are sorted by increasing number of edges deleted in the planar subgraph. We observe the effectiveness of postprocessing, underlining again that postprocessing is essential. We can also see that the *multi* variants come quite close to the corresponding *var* variants. This is again accompanied by a much better runtime of *multi*; in this case *multi-inc* is about 15–30 times faster than *var-inc*, and *multi-all* even about 100 times faster.

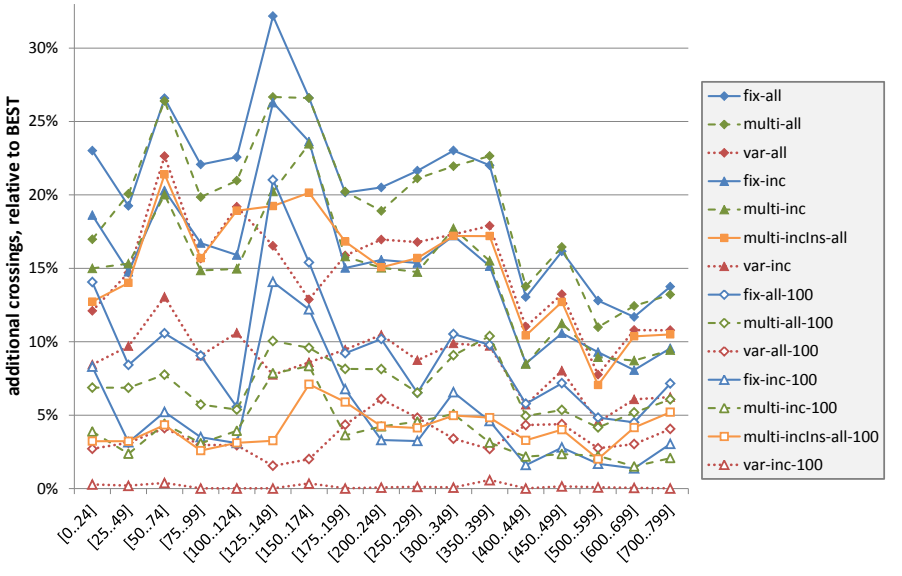


Fig. 5. Number of crossings for *AT&T* graphs, relative to best found solution

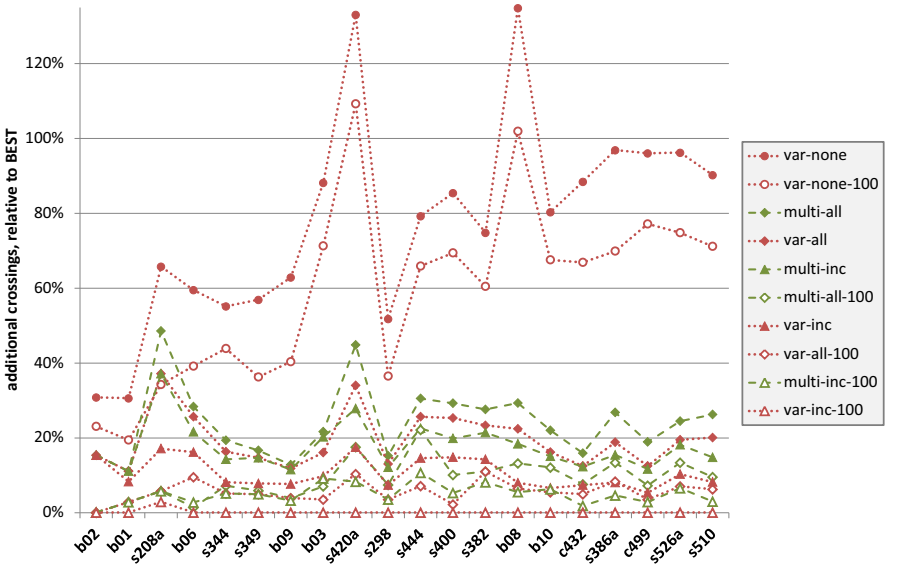


Fig. 6. Number of crossings for *ISCA* graphs, relative to best found solution

Conclusions. We presented *inc*, a new practically dominating postprocessing strategy for the planarization heuristic, and report on *multi*, the first implementation of any crossing minimization approximation algorithm for general graphs.

Both algorithms outperform any previously known heuristic in terms of solution quality, and if one cannot afford the relatively long running times for inserting all edges iteratively into a variable embedding using *inc*, the *multi* variants give the probably best balance between running time and solution quality: while being much faster, its solutions tend to be only slightly weaker than *inc*'s.

References

1. Batini, C., Talamo, M., Tamassia, R.: Computer aided layout of entity relationship diagrams. J. Syst. Software 4, 163–173 (1984)
2. Cabello, S., Mohar, B.: Crossing and Weighted Crossing Number of Near Planar Graphs. In: Tollis, I.G., Patrignani, M. (eds.) GD 2008. LNCS, vol. 5417, pp. 38–49. Springer, Heidelberg (2009)
3. Chimani, M.: Computing Crossing Numbers. PhD thesis, TU Dortmund, Germany (2008)
4. Chimani, M., Gutwenger, C., Mutzel, P., Wolf, C.: Inserting a vertex into a planar graph. In: Mathiru, C. (ed.) Proc. SODA 2009, pp. 375–383 (2009)
5. Chimani, M., Hliněný, P.: A tighter Insertion-Based Approximation of the Crossing Number. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6755, pp. 122–134. Springer, Heidelberg (2011)
6. Chimani, M., Hliněný, P., Mutzel, P.: Vertex insertion approximates the crossing number for apex. Europ. J. Comb. (to appear, 2011)
7. Chimani, M., Mutzel, P., Bomze, I.: A New Approach to Exact Crossing Minimization. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 284–296. Springer, Heidelberg (2008)
8. Chuzhoy, J., Makarychev, Y., Sidiropoulos, A.: On graph crossing number and edge planarization. In: Proc. SODA 2011, pp. 1050–1069. ACM Press (2011)
9. Di Battista, G., Garg, A., Liotta, G., Tamassia, R., Tassinari, E., Vargiu, F.: An experimental comparison of four graph drawing algorithms. Computational Geometry 7(5-6), 303–326 (1997)
10. Di Battista, G., Tamassia, R.: On-line planarity testing. SIAM Journal on Computing 25, 956–997 (1996)
11. Gutwenger, C.: Application of SPQR-Trees in the Planarization Approach for Drawing Graphs. PhD thesis, TU Dortmund, Germany (2010)
12. Gutwenger, C., Mutzel, P.: A Linear Time Implementation of SPQR Trees. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 77–90. Springer, Heidelberg (2001)
13. Gutwenger, C., Mutzel, P.: An Experimental Study of Crossing Minimization Heuristics. In: Liotta, G. (ed.) GD 2003. LNCS, vol. 2912, pp. 13–24. Springer, Heidelberg (2004)
14. Gutwenger, C., Mutzel, P., Weiskircher, R.: Inserting an edge into a planar graph. Algorithmica 41(4), 289–308 (2005)
15. Hliněný, P., Salazar, G.: On the Crossing Number of Almost Planar Graphs. In: Kaufmann, M., Wagner, D. (eds.) GD 2006. LNCS, vol. 4372, pp. 162–173. Springer, Heidelberg (2007)
16. Hopcroft, J.E., Tarjan, R.E.: Dividing a graph into triconnected components. SIAM Journal on Computing 2(3), 135–158 (1973)
17. Jünger, M., Leipert, S., Mutzel, P.: A note on computing a maximal planar subgraph using PQ-trees. IEEE Trans. Comp.-Aided Design 17(7), 609–612 (1998)
18. Ziegler, T.: Crossing Minimization in Automatic Graph Drawing. PhD thesis, Saarland University, Germany (2001)