Inserting an Edge Into a Planar Graph

Carsten Gutwenger* Petra Mutzel† René Weiskircher‡

Abstract

Computing a crossing minimum drawing of a given planar graph G augmented by an additional edge e in which all crossings involve e, has been a long standing open problem in graph drawing. Alternatively, the problem can be stated as finding a planar combinatorial embedding of a planar graph G in which the given edge e can be inserted with the minimum number of crossings. Many problems concerned with the optimization over the set of all combinatorial embeddings of a planar graph turned out to be NP-hard. Surprisingly, we found a conceptually simple linear time algorithm based on SPQR-trees, which is able to find a crossing minimum solution.

1 Introduction

Crossing minimization is among the most challenging problems in graph theory and graph drawing. Although, there is a vast amount of literature on this NP-hard problem (for a survey see, e.g., [13], NP-hardness is shown in [5]), so far no practically efficient exact algorithm for crossing minimization is known. Currently, the best known approach for crossing minimization is based on planarization. Here, in a first step, the minimum number of edges is deleted so that the resulting graph is planar. Then, the edges are iteratively reinserted into the planar subgraph so that the number of crossings is minimized. So far, this is done in the following way: Fix an arbitrary combinatorial embedding Π of the planar subgraph P and reinsert the first deleted edge e_1. This is done by solving a shortest path problem in the augmented (geometrical) dual graph of P associated with Π, since every crossing of e_1 and an edge f corresponds to using an edge in the dual graph. Then, the crossings are substituted by artificial vertices, yielding a planar graph again. Now, the next edge can be inserted, and so on.

One criticism of the planarization method was that when choosing a "bad" embedding in the edge reinsertion phase, the number of crossings may get much higher than necessary [8]. Hence, the question arose if there is a polynomial time algorithm for inserting an edge into the planar subgraph P so that the number of crossings is minimized. Thereby, the task is to optimize over the set of all possible combinatorial embeddings of P.

While it is possible to compute an arbitrary combinatorial embedding for a planar graph in linear time [10, 4], it is often hard to optimize over the set of all possible combinatorial embeddings. E.g., the problem of bend minimization can be solved in polynomial time for a fixed combinatorial embedding [14], while it is NP-hard over the set of all combinatorial embeddings [6]. When a linear function of polynomial size is defined on the cycles of a graph, it is NP-hard to find the embedding that maximizes the value of the cycles that are face cycles in the embedding [12, 11]. Note that the number of combinatorial embeddings of a planar graph may be exponential.

This paper shows that the edge reinsertion problem can be solved in polynomial time, thus solving a long standing open problem in graph drawing. We present a conceptually simple linear time algorithm based on SPQR-trees which is able to solve the edge reinsertion problem to optimality.

*Max–Planck–Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

†Technische Universität Wien E186, Favoritenstraße 9-11, 1040 Wien, Austria

‡Technische Universität Wien E186, Favoritenstraße 9-11, 1040 Wien, Austria

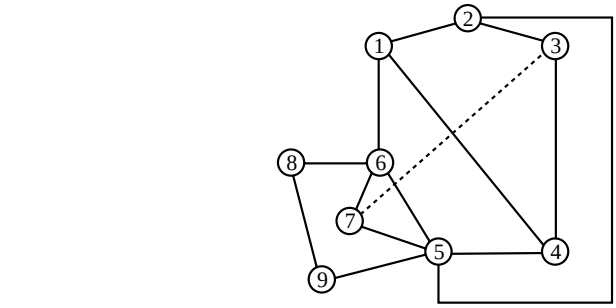
A graph is *planar* if it can be drawn in the plane without any edge crossings. *(Combinatorial) embeddings* are equivalence classes of planar drawings which can be defined by the sequence of the edges around each vertex in a drawing. We consider two drawings of the same graph equivalent, if the circular sequences of the adjacent edges around each node in clockwise order is the same. We say that they realize the same combinatorial embedding.

Figure 1 shows a simple case where the choice of the combinatorial embedding of the planar subgraph has an impact on the number of crossings produced when inserting the dashed edge. When choosing the embedding of Figure 1(a) for the planar subgraph (without the dashed edge), we get two crossings, while the optimal crossing number over the set of all combinatorial embeddings is one (see Fig. 1(b)).

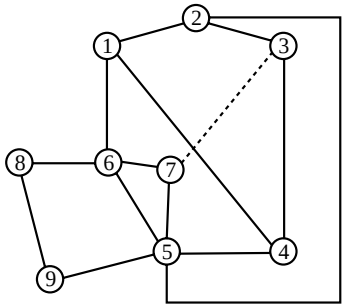
Formally, we define the edge insertion problem as follows: Given a planar graph $G = (V, E)$ and a pair of vertices (v_1, v_2) in G, find an embedding Π of G such that we can add the edge $e = (v_1, v_2)$ to Π with the minimum possible number of crossings among all embeddings of G. We will present an algorithm that is able to solve the problem in linear time.

Note that the solution produced by our algorithm does not necessarily lead to a drawing of the graph $G' = (V, E \cup \{e\})$ with the minimum number of crossings. The reason is that there may not always be a drawing with the minimum number of crossings in which a maximum planar subgraph is drawn without crossings.

In Section 2, we give a brief overview of SPQR-trees and define a few concepts we need. Section 3 contains the algorithm for solving our problem for biconnected graphs and mentions briefly the extension of the algorithm for general planar graphs. We also prove the correctness and discuss the running time. Section 4 gives our computational experiments on a set of benchmark graphs. They show that our algorithm produces significant improvements compared with the state-of-the-art insertion method. The last section addresses an interesting open problem.



(a)



(b)

Figure 1: The number of crossings when inserting an edge highly depends on the chosen embedding

2 Preliminaries

In this section, we give a brief overview of the SPQR-tree data structure for biconnected graphs. A connected graph is biconnected if it contains no vertex whose removal splits the graph into two or more components. SPQR-trees have been suggested by Di Battista and Tamassia [3]. They represent a decomposition of a biconnected graph into triconnected components. A connected graph is triconnected, if there is no pair of vertices in the graph whose removal splits the graph into two or more components.

The structure of the SPQR-tree T for a graph G is determined by the *split pairs* of G. These are

pairs of vertices that are either connected by an edge or whose removal splits the graph into components. These components are called the *split components* of the split pair.

An SPQR-tree has four types of nodes and with each node is associated a biconnected graph which is called the *skeleton* of that node. This graph can be seen as a simplified version of the original graph and its vertices are vertices of the original graph. If (u, v) is and edge in a skeleton of a node in the SPQR-tree T of a graph G, then the vertices u and v are a split pair of G and the edge (u, v) represents one or several split components of the split pair (u, v).

1. Q-**node:** The skeleton consists of two vertices connected by two edges. One of the edges represents an edge of the original graph and the other one the rest of the graph. There is exactly one Q-node for each edge in the graph and these nodes are the leaves of the SPQR-tree (the nodes with degree one).

2. S-**node:** The skeleton is a simple cycle with at least 3 vertices (see Fig. 2(a)).

3. P-**node:** The skeleton consists of two vertices connected by at least three edges (see Fig. 2(b)).

4. R-**node:** The skeleton is a triconnected graph with at least four vertices (see Fig. 2(c)).

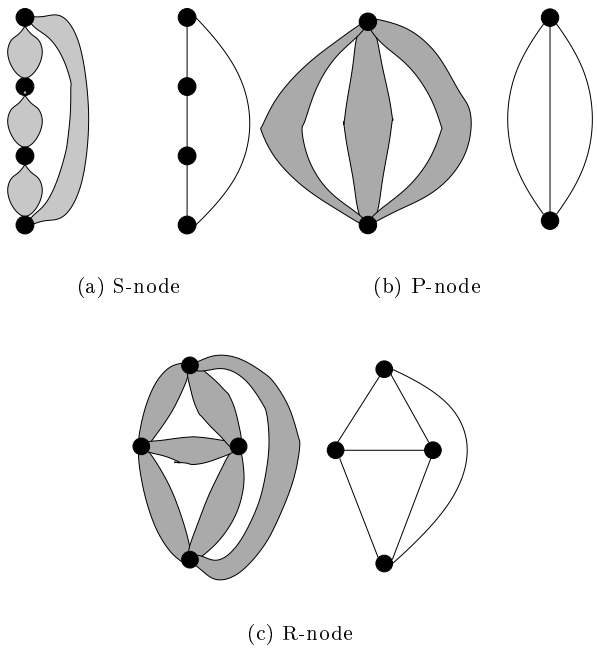


Figure 2: The skeletons of the three inner node types of SPQR-trees and how there edges correspond to subgraphs

Except for the edges in the skeletons of Q-nodes that represent an edge of the original graph, all edges in the skeletons correspond to exactly one edge in the SPQR-tree and each edge in the SPQR-tree corresponds to exactly one edge in each of the the two skeletons of the nodes that it connects. These two edges in the two skeletons of adjacent nodes are called *twin edges* because they correspond to the same edge in the SPQR-tree and they connect the same vertices. As a consequence, every vertex of the original graph is contained in at least three skeletons (it is contained in at least two Q-node skeletons because every vertex in a biconnected graph has at least degree two and in at least one skeleton of an inner node of the SPQR-tree). For each vertex v in the original graph, we call all nodes in the SPQR-tree whose skeletons contain v the *allocation nodes* of v. They will play an important role in our algorithm.

When we see the SPQR-tree as an unrooted tree, then it is unique for each biconnected planar graph. Another important property of these trees is that their size (including the skeletons) is linear in the size of the original graph and that they can be constructed in linear time [7].

As described in [3], SPQR-trees can be used to represent all combinatorial embeddings of a biconnected planar graph. This is done by choosing embeddings for the skeletons of the nodes in the tree. The skeletons of S- and Q-nodes are simple cycles, so they have only one embedding. Therefore, we only have to look at the skeletons of R- and P-nodes. The skeletons of R-nodes are triconnected graphs. Our definition of combinatorial embeddings distinguishes between two combinatorial embeddings of a tricon-

nected graph, which are mirror-images of each other (the circular order of the edges around each vertex in clockwise order is reversed in the second embedding). The number of different embeddings of a P-node skeleton is $(k-1)!$ where k is the number of edges in the skeleton.

Every combinatorial embedding of the original graph defines a unique combinatorial embedding for each skeleton of a node in the SPQR-tree. Conversely, when we define an embedding for each skeleton of a node in the SPQR-tree, we define a unique embedding for the original graph.

In this paper, we describe an algorithm for solving the following problem: Given a planar graph G and two non-adjacent vertices, find an optimal *edge insertion path* and a corresponding embedding Π of G for this path. So first we have to define what is meant by the term edge insertion path.

Definition 1 *Let v_1 and v_2 be two non-adjacent vertices in a planar graph G and Π a combinatorial embedding of G. Let $\overline{G_\Pi}$ be the dual graph of G with respect to Π. By $\bar{e}$, we denote the edge in $\overline{G_\Pi}$ corresponding to edge e in G. With $\bar{f}$ we denote the vertex in $\overline{G_\Pi}$ corresponding to face f in Π. Then the list $L = (e_1, \ldots, e_k)$ of edges in G is an* edge insertion path *for v_1 and v_2 in G with respect to Π if the following conditions are satisfied:*

1. *There is a face in Π with e_1 and v_1 on its boundary.*
2. *There is a face in Π with e_k and v_2 on its boundary.*
3. *$\bar{L} = (\overline{e_1}, \ldots, \overline{e_k})$ represents a path in $\overline{G_\Pi}$.*

This basically means that we can add the edge (v_1, v_2) to Π with k crossings, each involving edge e_i for $1 \leq i \leq k$ and the edge (v_1, v_2). We call an edge insertion path an *optimal edge insertion path* for v_1 and v_2, when there is no shorter edge insertion path for v_1 and v_2 with respect to any combinatorial embedding of the graph.

Figure 3 shows three different edge insertion paths for v_1 and v_2 with respect to the embedding realized by the drawing. The three paths are the empty path, the path (e_1, e_2, e_3) and the path (e_4, e_5, e_6). In this case, the empty path is the only optimal edge insertion path for v_1 and v_2.

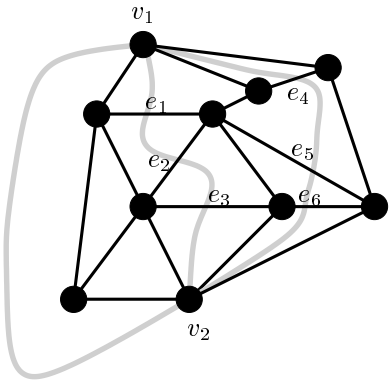


Figure 3: Three different edge insertion paths for v_1 and v_2

In the description of our algorithm we will need the notation of the *expansion graph* of an edge $e = \{v_1, v_2\}$ in a skeleton of a node v in an SPQR-tree T. As already stated, each skeleton edge is associated with a tree edge. We can produce the expansion graph of e as follows: We remove the tree edge $e_T = (v, w)$ associated with e from T splitting T into two trees T_1 (the part containing v) and T_2. Now we transform T_2 into an SPQR-tree T_2' by adding a Q-node to T_2 adjacent to w. This new Q-node represents e. The biconnected graph whose SPQR-tree is T_2' is called the expansion graph of e.

3 The algorithm

We will first present the algorithm for biconnected graphs, proof its correctness and optimality and then briefly mention its extension for connected planar graphs and general planar graphs. One term we will use in our algorithm is the *augmentation* of a dual graph with two vertices.

Definition 2 *Let v_1 and v_2 be two vertices in the planar graph G and Π be an embedding of G. Let $\overline{G_\Pi}$ be the dual graph of G with respect to embedding*

Π. *We say the graph* $\overline{G_\Pi}'$ *has been obtained by* augmenting $\overline{G_\Pi}$ with v_1 and v_2, *if it was constructed by adding* v_1 *and* v_2 *to* $\overline{G_\Pi}$ *and inserting edges from* v_i *to all vertices in* $\overline{G_\Pi}$ *representing faces in* Π *with* v_i *on their boundary for* $i \in \{1, 2\}$.

Algorithm 1 computes an optimal edge insertion path for a biconnected planar graph and two non-adjacent vertices in this graph. As already mentioned in section 2, the term *allocation node* of a vertex v in a graph G used in the algorithm refers to a node in the SPQR-tree T of G with v in its skeleton. We say an edge e in a skeleton *represents* a vertex v of G if v is contained in its expansion graph. By *expanding* an edge e in a skeleton, we mean replacing this edge by its expansion graph where we have removed the edge e. Expanding all edges in a skeleton results in the original graph.

We use the fact that any embedding of G is uniquely defined by the embedding of every skeleton in T. Our algorithm solves the problem of finding an optimal edge insertion path for v_1 and v_2 in G by finding optimal edge insertion paths in the skeletons of certain nodes in the SPQR-tree T of G where we expanded certain edges and then concatenating them. We only have to do this for nodes of T that lie on the shortest path P in T connecting an allocation node of v_1 and v_2. Note that this path is uniquely defined because T is a tree.

We don't have to consider the nodes in the SPQR-tree that are not on P because their embedding does not influence the length of a shortest edge insertion path. If we look at two embeddings Π_1 and Π_2 of G that only differ in the embedding of the SPQR-nodes that are not on P, then for every optimal edge insertion path for v_1 and v_2 with respect to Π_1 there exists an optimal edge insertion path for v_1 and v_2 with respect to Π_2 that contains the same edges (though the sequence may differ) and vice versa. This is shown in the proof of theorem 2.

For each of the nodes v on P with skeleton S, each of the vertices v_1 and v_2 is either present in S or represented by an edge. If the latter is the case, we split this edge by inserting an artificial vertex. So there will always be a representative for v_1 and v_2 in the resulting graph. Then we compute an optimal edge

Algorithm 1: Algorithm `OptimalBlockInserter` for computing an optimal edge insertion path for a pair of non-adjacent nodes in a biconnected planar graph

Input: A biconnected planar graph G, and two non-adjacent vertices v_1 and v_2 in G.
Result: An optimal edge insertion path L for v_1 and v_2 with respect to some embedding Π of G

begin
- Compute the SPQR-tree T of G;
- $L \leftarrow ()$;
- Determine arbitrary allocation nodes μ_1 of v_1 and μ_2 of v_2;
- Find the path P_1 in T starting at μ_1 and ending at μ_2;
- Delete nodes from the start and end of P_1 until we have produced the shortest path P_2 in T from an allocation node of v_1 to an allocation node of v_2;
- Delete all nodes from P_2 except the R-nodes, producing the list P_3 of nodes in T;
- **while** P_3 *is not empty* **do**
  - Pop the first node v from P_3 and let S be its skeleton;
  - **if** v_1 *is in* S **then**
    - $x_1 \leftarrow v_1$;
  - **else**
    - Split the edge representing v_1 in S by inserting a new vertex y_1;
    - Mark the two edges produced by the split;
    - $x_1 \leftarrow y_1$;
  - **if** v_2 *is in* S **then**
    - $x_2 \leftarrow v_2$;
  - **else**
    - Split the edge representing v_2 in S by inserting a new vertex y_2;
    - Mark the two edges produced by the split;
    - $x_2 \leftarrow y_2$;
  - Expand all unmarked edges in S;
  - Compute an arbitrary embedding Π for S;
  - Compute the dual graph $\overline{S_\Pi}$;
  - Augment $\overline{S_\Pi}$ with x_1 and x_2;
  - Compute the shortest path L' in $\overline{S_\Pi}$ from x_1 to x_2;
  - Delete the first and the last edge in L';
  - Replace every dual edge in L' by its primal counterpart;
  - Append L' to L;

end

insertion path connecting the two representatives in the graph that we get by expanding all edges in the skeleton except the ones that we have split. It is not hard to see that such an optimal edge insertion path will always be empty if v is a P- or S-node. So our algorithm only has to deal with the R-nodes on P.

Computing such an optimal edge insertion path in an R-node skeleton where edges have been expanded is not hard since the skeletons of R-nodes are triconnected graphs. Therefore, they have only two different combinatorial embeddings that are mirror images of each other. If we compute an optimal edge insertion path for one embedding, it will also be an optimal edge insertion path for the other embedding. We can compute the edge insertion path for a fixed embedding using a shortest path algorithm on the dual graph. The embedding we choose for the expansion graphs does not matter for the length of a shortest edge insertion path. This is shown in the proof of theorem 2.

Algorithm 1 only computes an edge insertion path L for the two vertices v_1 and v_2, but there is a simple way for finding an embedding Π such that L is an edge insertion path for v_1 and v_2 with respect to Π. We just split every edge in L by introducing new vertices and connect these vertices with edges to form a path starting at v_1 and ending at v_2. Since L is an edge insertion path, the graph G' generated by this operation is planar. Therefore, we can compute a combinatorial embedding Π' for G' in linear time. When we replace all the split edges in Π' with the original edges, we get a combinatorial embedding Π for G with the property that L is an edge insertion path for v_1 and v_2 with respect to Π.

We show correctness of the algorithm in two parts.

Theorem 1 *Given a planar biconnected graph G and two non-adjacent vertices v_1 and v_2 in G, Algorithm 1 computes a list of edges L such that L represents an edge insertion path for v_1 and v_2 with respect to some embedding Π of G.*

Proof (sketch) The proof centers on the nodes of T on path P_2. The reason for not using P_3 (the set of nodes that the algorithm works on) is that the task of the algorithm is to compute the edge insertion path L, while in this proof, we have to compute an embedding Π and show that the edge insertion path computed by the algorithm is valid for the embedding. To compute the edge insertion path, we don't have to look at all the skeletons in T but if we want to define an embedding, we have to define an embedding for every skeleton.

Let P_2 be given by the sequence $P_2 = (p_1, \ldots, p_k)$ of nodes in T. p_1 is an allocation node of v_1 and p_k is an allocation node of v_2. So the skeleton of p_1 contains v_1 and the skeleton of p_k contains v_2 (note that p_1 and p_k might be identical). By the construction of P_2, none of the nodes p_i with $1 < i < k$ has the vertex v_1 or v_2 in its skeleton.

The case $k = 1$ is not hard to verify, so assume $k > 1$. For $i = 1, \ldots, k-1$, let G_i be the graph obtained from the skeleton of p_i by expanding all skeleton edges except for the representative e_2 of v_2 and then replacing e_2 by a path of length two linked by the new node r_i. Furthermore, let $G_k = G$ and $r_k = v_2$. We will show by induction that for each $i \in \{1, \ldots, k\}$ there is a combinatorial embedding Π_i of G_i such that the edges in L which are also contained in G_i form a prefix L_i of L and L_i is an edge insertion path for v_1 and r_i with respect to Π_i. Thus, Π_k is an embedding for G and $L_k = L$ is an edge insertion path for G with respect to Π_k. So Π_k is the embedding Π of the theorem.

We construct Π_k iteratively. The construction is done in k stages. We build graph G_i for $i \in \{1, \ldots, k\}$ and for each of these graphs we construct an embedding Π_i. Each graph G_i for $1 \leq i < k$ can be constructed from G_{i+1} by replacing a subgraph with a path of two edges. The inner vertex of this path is r_i.

We prove that for all $i \in \{1, \ldots, k-1\}$, the prefix L_i of L with the property that all edges of L_i are contained in G_i is an edge insertion path for v_1 and r_i with respect to Π_i and that the suffix of L following L_i does not contain any edges from G_i. Each L_i with $i \in \{1, \ldots, k-1\}$ is a prefix of L_{i+1} (see Fig. 4).

We only show the induction step: Assume that we have already constructed the graph G_{l-1} and the embedding Π_{l-1} and we want to construct graph G_l and the embedding Π_l. To do this, we start with the skeleton of p_l and construct the graph G'_l by expanding all the skeleton edges whose expansion graph contains

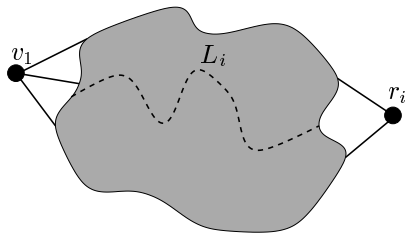


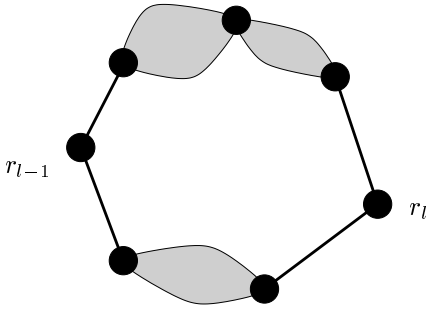
Figure 4: The graph G_i of stage i in the proof for Theorem 1

neither v_1 nor v_2 (if we expanded all the edges, we would produce the original graph G). The edge e_1 whose expansion graph contains v_1 forms the link to G_{l-1} and the edge e_2 with v_2 in the expansion graph forms the link to G'_{l+1} (in the case $l = k$, e_2 does not exist because v_2 is a vertex in the skeleton of p_k).

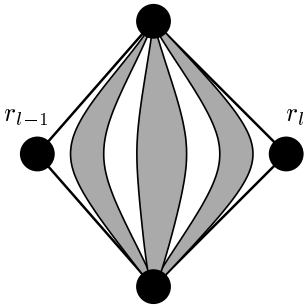
We replace e_1 with a path of two edges connected by vertex r_{l-1} and, if $l < k$, e_2 with a path of two edges connected by vertex r_l. We claim that the subsequence S of L containing only edges in G'_l is an edge insertion path for r_{l-1} and r_l with respect to some embedding Π'_l of G'_l. If p_l is an S- or P-node, Algorithm 1 has not added any edges contained in G'_l to L, so S is empty. In the first case, the removal of r_{l-1} and r_l will disconnect G'_l and so any embedding of G'_l has the property that we can insert the edge (r_{l-1}, r_l) without crossings (see Fig. 5(a)). In the second case, it is easy to construct an embedding in which r_{l-1} and r_l are on the boundary of the same face (see Fig 5(b)). If p_l is an R-node, we have computed an edge insertion path for r_{l-1} and r_l in Algorithm 1 and this edge insertion path works for both embeddings of G'_l because they are mirror images of each other.

Now we need to combine the embedding Π'_l with the embedding Π_{l-1} computed in the previous steps to produce embedding Π_l of G_l. When we do this correctly (omitted), we can afterwards guarantee that a prefix L_l of L is an edge insertion path for v_1 and r_l in G_l with respect to embedding Π_l. □

Theorem 2 *Algorithm 1 computes an optimal edge insertion path for v_1 and v_2.*



(a)



(b)

Figure 5: The cases where p_l is an S- or P-node

Proof (sketch) We have already seen that Algorithm 1 computes an edge insertion path for v_1 and v_2, so let L be the computed path with respect to embedding Π. We need to show that each edge insertion path for v_1 and v_2 has length at least $|L|$. Let L' be an arbitrary edge insertion path with respect to some embedding Π'.

Let G_i be the graph we get by expanding all edges in the skeleton of node p_i on P_2 except the representatives of v_1 and v_2. If we assume $|L'| < |L|$, then there must be at least one R-node p_l on P_2 with the property that the path P' in the dual of G_l defined by L' is shorter then the path P defined by L.

We consider the skeleton S_l of p_l. We have computed G_l by expanding all edges except for the ones

whose expansion graph contains v_1 or v_2 (the representatives of v_1 and v_2) and replacing the representatives with paths of two edges linked by the vertices r_{l-1} an r_l. The important observation is that the embedding we choose for the expansion graphs has no influence on the length of a shortest edge insertion path for r_{l-1} and r_l.

Intuitively this can be seen as follows: An edge insertion path P in G_l defines an edge insertion path P_S in S_l. When P_S does not include some edge e of the skeleton, the embedding of the expansion graph $X(e)$ of e is obviously irrelevant, because P does not cross any of the edges in $X(e)$. Now we assume that P_S includes e. So P_S connects the face F_1 left of e with the face F_2 right of e by crossing e. In a corresponding embedding of G_l, P must connect the face left of $X(e)$ with the face right of $X(e)$. Therefore it must traverse the graph $X(e)$. An optimal path for traversing $X(e)$ will produce the same number of crossings no matter how $X(e)$ is embedded, only the sequence of the crossed edges changes. This can be shown by structural induction on the SPQR-tree.

An embedding of G_l is determined by the embedding of S_l and the embeddings of the expansion graphs. The embeddings of the expansion graphs are irrelevant and there are only two embeddings of S_l which are mirror images of each other. Therefore, we can choose an arbitrary embedding of G_l to compute the edge insertion path. So there can be no shorter edge insertion path P' for r_{l-1} and r_l in G_l and our algorithm must compute an optimal edge insertion path. □

The linear running time is not hard to see: A planar graph with n vertices has at most $3n - 6$ edges (if we assume that there are no multi-edge and no self loops). We can compute an SPQR-tree in linear time and the size of the tree including the skeletons is also linear. Thus we can compute the paths P_1, P_2 and P_3 in linear time. Two graphs G_1 and G_2 computed in the while-loop of Algorithm 1 share at most two vertices and are (except for a constant number of nodes and edges) subgraphs of the original graph. Computing arbitrary embeddings and computing a shortest path in the dual graph with breadth first search can be done in linear time so the running time of the while-loop is linear in the size of the original graph.

To deal with connected planar graphs, we use algorithm 1 as a subroutine. We use a data structure called the *block tree*. This tree has two types of nodes: The $B-$nodes correspond to biconnected components of the original graph G and the $V-$nodes to the vertices. The $B-$nodes contain the SPQR-trees of the biconnected component they represent. There are only edges between $V-$ and $B-$nodes. An edge is present between a $V-$node and a $B-$node, if the corresponding vertex is contained in the component represented by the $B-$node.

Our algorithm first computes the path P in the block tree B of G connecting v_1 and v_2. For each $B-$node b on this path, the two representatives of v_1 and v_2 are computed. The representative of vertex v_i for $i \in \{1, 2\}$ is either v_i itself if it is contained in the biconnected component c represented by b or the vertex represented by the next $V-$node on the path from b to v_i. Using algorithm 1, an optimal edge insertion path for v_1 and v_2 in c is computed. The result of the algorithm is the concatenation of all the edge insertion paths computed for all $B-$nodes on P.

In the proof of the correctness of the algorithm (which is omitted here because of space considerations), we first show that an optimal edge insertion path for v_1 and v_2 will not cross any edge in a biconnected component of G that is not represented by a $B-$node on P. We show that for any edge insertion path that crosses edges of components not represented on P, we can construct a shorter edge insertion path by deleting a subpath of the edge insertion path. Using the correctness and optimality of algorithm 1, we then proof by contradiction that there can be no shorter edge insertion path for v_1 and v_2 in G than the one computed by our algorithm.

It is easy to extend the algorithm to planar graphs that are not connected. If the two vertices we want to connect by an edge are contained in the same connected component, we can use the algorithm for connected graphs. Otherwise, we can always connect the edges without introducing a crossing.

4 Computational experiments

We have implemented our algorithm using AGD [1], a library of algorithms for graph drawing, which contains a state-of-the-art implementation of crossing minimization using planarization and a linear time implementation of SPQR-trees [7]. In our tests, we use the 8249 non-planar graphs from a benchmark set collected by Di Battista et al. [2] ranging from 10 to 100 vertices. The planar subgraph is computed using the AGD implementation of [9]. In 10.1% (831 graphs) of our benchmark graphs the planar subgraphs resulted from deleting a single edge. In these instances, our algorithm is direct applicable. In the remaining cases, we insert the edges iteratively (see Section 1) applying our new algorithm. It is not hard to show that the resulting graphs G_i after every iteration i satisfy the following property: In every combinatorial embedding of G_i all artificial vertices represent real crossings.

We compare the number of crossings produced by our approach with the number of crossings produced by the standard algorithm described in the introduction. Notice that, since we insert the edges iteratively in the second phase of the planarization method, the impact of optimally inserting one edge is not obvious.

Figure 6 visualizes the tremendous improvement achieved by our new algorithm. It displays for each graph the relative improvement I_R of the number of crossings in percent. Let c_s denote the number of crossings by the standard algorithm and c_n denote our new edge insertion algorithm, then $I_R = \frac{c_s - c_n}{c_s} 100\%$.

For 68% out of the 8249 tested non-planar graphs, the crossing number was smaller using our new method and for only 8% it was greater. The average relative improvement was 14.42% in total. The maximum improvement was 85.71%. This happened for a graph with 39 vertices and 56 edges. The standard algorithm produced 7 crossings whereas our new algorithm only 1. The maximum negative improvement was -100% for a graph with 65 vertices and 76 edges. Here, the standard algorithm produced 2 crossings and the new algorithm 4 crossings. The average number of crossings produced by the standard algorithm grows from 1.29 for graphs with 11 vertices to 57.86 for graphs with 100 vertices, whereas the numbers grow from 1.29 to 49.83 for our new algorithm.

The two figures 7 and 8 show the running time of the reinsertion step of the planarization method when the algorithm presented in this paper is used. In Fig. 7, we computed the average time needed for reinserting all edges for all graphs with the same number of vertices, while in Fig. 8, we computed the average time needed for the reinsertion step for all graphs where the same number of edges had to be reinserted. It seems that the time needed for the reinsertion step grows cubic with the number of nodes and almost quadratic with the number of edges that have to be reinserted.

The reason for these two facts is that the number of edges we have to reinsert grows roughly linear with the number of vertices in the graphs of our test suit and that whenever we reinsert one edge, the reinsertion of the next edge takes more time because the graph where we have to reinsert the edge is larger than before. Each reinsertion adds at least two edges and one vertex to the graph. The reason is that we can not reinsert the edge without producing at least one crossing, so we have to add at least one artificial vertex to make the graph planar again. So the insertion of a single edge is done in linear time with respect to the size of the graph where its is inserted, but the graph grows linearly with every reinserted edge.

5 Open problems

Our algorithm computes the optimum embedding of a planar graph for inserting an additional edge with respect to the number of crossings. It may be interesting to explore the connection of this problem with the general crossing minimization problem.

An interesting question is the following: Consider a graph with skewness one, i.e., a non-planar graph that can be made planar by deleting a single edge. All maximum planar subgraphs of the graph can be found in quadratic time by testing planarity of each subgraph constructed by deleting one edge. If we take the maximum planar subgraphs and apply our algo-

rithm to reinsert the deleted edge with the minimum number of crossings, do we always produce a drawing of the original graph with the minimum number of crossings?

References

[1] *AGD User Manual (Version 1.1)*, 1999. Universität Wien, Max-Planck-Institut Saarbrücken, Universität Trier, Universität zu Köln. See also `http://www.mpi-sb.mpg.de/AGD/`.

[2] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7:303–326, 1997.

[3] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996.

[4] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *J. of Computer and System Sciences*, 30(1):54–76, 1985.

[5] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal Alg. Disc. Methods*, 4:312–316, 1983.

[6] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. *Lecture Notes in Computer Science*, 894:286–297, 1995.

[7] C. Gutwenger and P. Mutzel. A linear time implementation of spqr trees. In J. Marks, editor, *Graph Drawing (Proc. 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer-Verlag, 2001.

[8] D. Harel and M. Sardas. Randomized graph drawing with heavy duty preprocessing. *Advanced Visual Interfaces*, pages 19–33, 1994.

[9] M. Jünger, S. Leipert, and P. Mutzel. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design*, 17(7):609–612, 1998.

[10] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[11] P. Mutzel and R. Weiskircher. Optimizing over all combinatorial embeddings of a planar graph. In G. Cornuéjols, R. Burkard, and G. Wöginger, editors, *Proceedings of the Seventh Conference on Integer Programming and Combinatorial Optimization (IPCO)*, volume 1610 of *LNCS*, pages 361–376. Springer Verlag, 1999.

[12] P. Mutzel and R. Weiskircher. Computing optimal embeddings for planar graphs. In *Proceedings of the Sixth Annual International Computing and Combinatorics Conference (COCOON)*, LNCS, pages 95–104. Springer Verlag, 2000.

[13] F. Shahrokhi, L. A. Székely, and I. Vrtó. Crossing numbers of graphs, lower bound techniques and algorithms: a survey. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 131–142. DIMACS, Springer-Verlag, October 1994. ISBN 3-540-58950-3.

[14] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16(3):421–444, 1987.

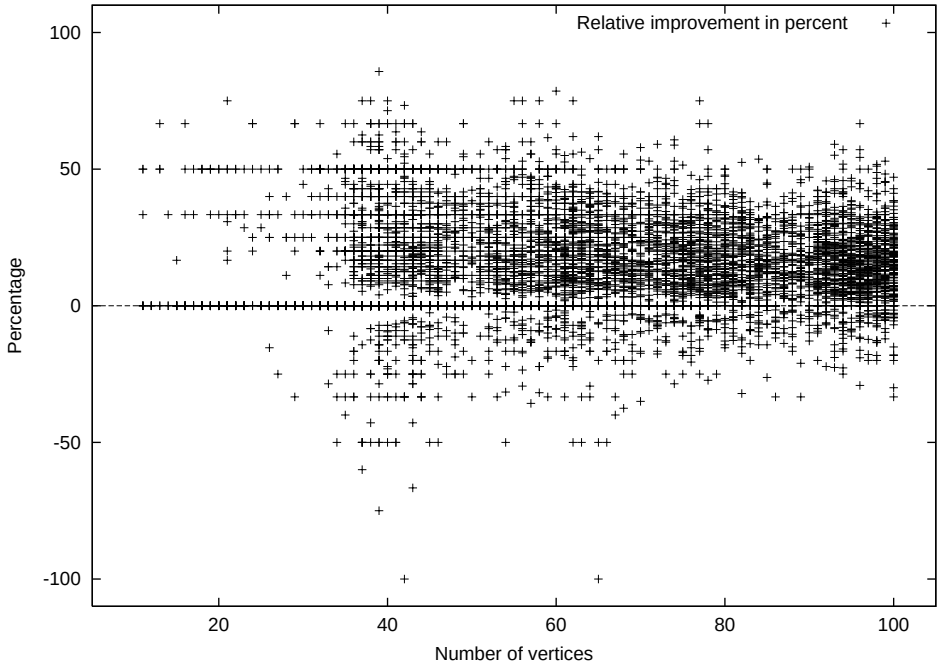


Figure 6: Relative improvement for each graph

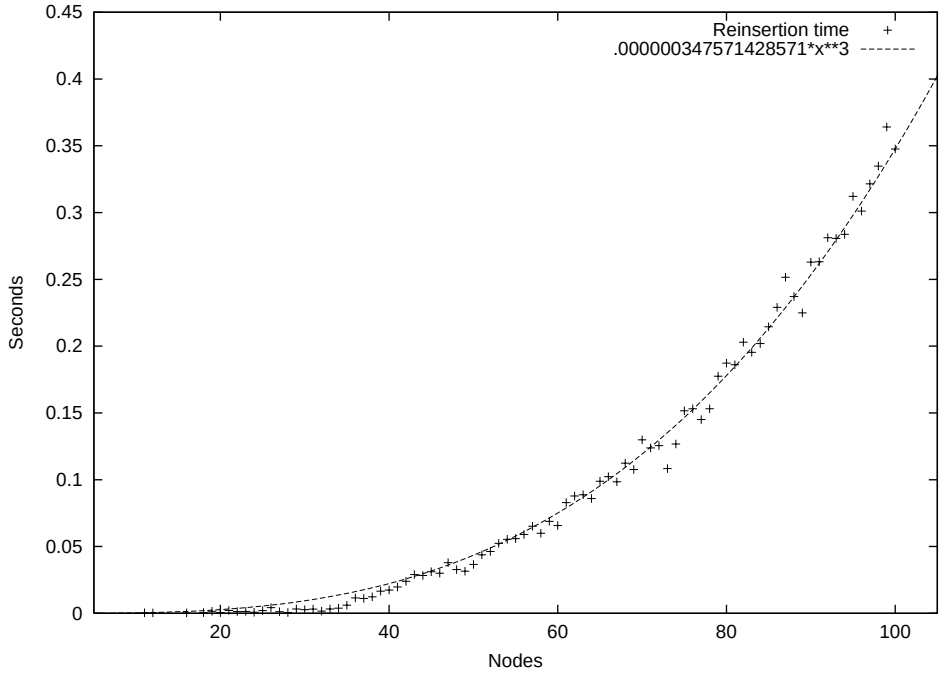


Figure 7: Average time needed for reinserting the edges in the planarization method using the optimal method for inserting an edge into a planar graph

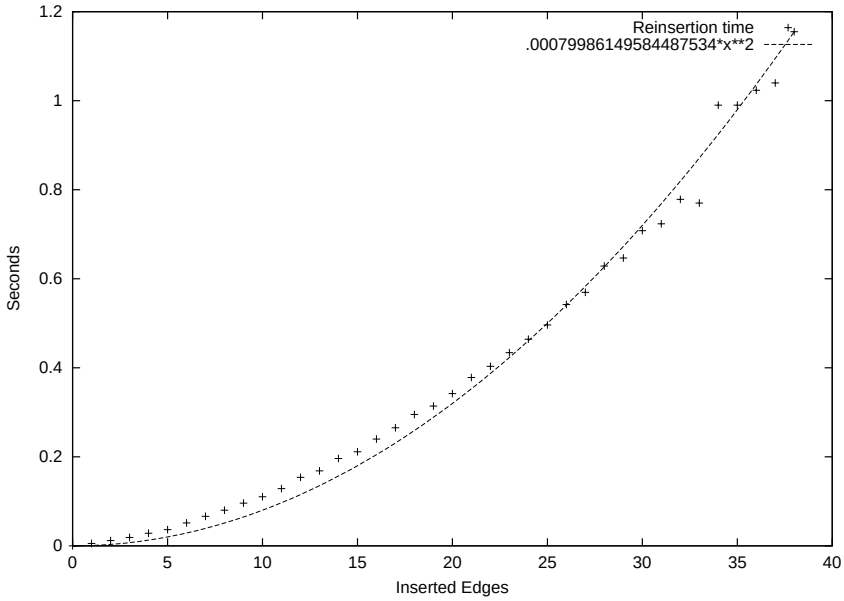


Figure 8: Average time needed for reinsertion when the average is computed for all graphs where the same number of edges have to be reinserted