UNIVERSITÄT DORTMUND

Efficient Extraction of Multiple Kuratowski Subdivisions (TR)

Markus Chimani
Petra Mutzel
Jens M. Schmidt

Algorithm Engineering Report
TR07-1-002
June 2007
ISSN 1864-4503

University of Dortmund
Department of Computer Science
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

Efficient Extraction of Multiple Kuratowski Subdivisions (TR)

Markus Chimani, Petra Mutzel, and Jens M. Schmidt

Department of Computer Science, University of Dortmund, Germany
{markus.chimani,petra.mutzel,jens.schmidt}@uni-dortmund.de

Technical Report TR07-1-002
June 2007

Abstract. A graph is planar if and only if it does not contain a Kuratowski subdivision. Hence such a subdivision can be used as a witness for non-planarity. Modern planarity testing algorithms allow to extract a single such witness in linear time. We present the first linear time algorithm which is able to extract multiple Kuratowski subdivisions at once. This is of particular interest for, e.g., Branch-and-Cut algorithms which require multiple such subdivisions to generate cut constraints. The algorithm is not only described theoretically, but we also present an experimental study of its implementation.

1 Introduction

A *planar drawing* of a graph is an injection of its vertices onto points in the plane, and a mapping of the edges into open curves between their endpoints. These curves are not allowed to touch each other, except in their common endpoints. Graphs which admit such a planar drawing, are called *planar graphs*, and recognizing this graph class has been a vivid research topic for the past decades. Hopcroft and Tarjan [12] showed in 1974 that this problem can be solved in linear time, using sophisticated data structures and intricate algorithms. Current planarity testing algorithms like the ones by Boyer and Myrvold [4,5] and de Fraysseix, Ossona de Mendez and Rosenstiehl [9,10,11] are less complex but still quite involved.

As shown by Kuratowski [15] in 1930, a graph is planar if and only if it does not contain a $K_{3,3}$ or a K_5 subdivision, i.e., a complete bipartite graph $K_{3,3}$ or complete graph K_5 with edges replaced by paths of length at least one. Such subgraphs are called *Kuratowski subdivisions.* The efficient extraction of such a witness of non-planarity was non-trivial in the context of the first linear planarity tests. Linear algorithms for such an extraction were later presented by Williamson [17] and Karabeg [14]. Modern planarity testing algorithms like the ones by Boyer and Myrvold, and de Fraysseix et al. can directly extract a single Kuratowski subdivision, if the given graph is non-planar.

In ILP-based Branch-and-Cut approaches which try to solve, e.g., the Maximum Planar Subgraph problem [13] or the Crossing Minimization problem [6,7], the identification of multiple such witnesses is a crucial part. Thereby, we look at some intermediate solution and try to find Kuratowski subdivisions. For each such subdivision, we can try to generate a cut constraint, necessary to efficiently solve the ILP. Experience shows that it is desirable to find multiple Kuratowski constraints at once, as they strengthen the LP-relaxation of the problem.

In the following, let $G = (V, E)$ be a non-planar undirected graph, without selfloops and multi-edges. Current planarity tests are able to extract a single Kuratowski subdivision in linear time $O(n)$, $n := |V|$. We address the problem of finding multiple Kuratowski subdivisions in efficient time. As there may exist exponentially many Kuratowski subdivisions in general, it is not practical to enumerate all of them. A basic approach would be to obtain k Kuratowski subdivisions through calling a planarity test k times and subsequently deleting an involved Kuratowski edge. This approach has a superlinear runtime of $O(kn)$, but we are not aware of any algorithm faster than this approach, up until now.

In this paper, we propose an algorithm which extracts multiple Kuratowski subdivisions in optimal time $O(n + m + \sum_{K \in \mathcal{S}} |E(K)|)$, with $\mathcal{S}$ being the set of identified Kuratowski subdivisions and $m := |E|$. This runtime is linear in the graph size and the extracted Kuratowski edges. The algorithm is based on the planarity test of Boyer and Myrvold [5] which is one of the fastest planarity tests today [3]. We will only give a short introduction into this planarity test in Section 2; for a full description of the original test see [5]. The main part of this paper focuses on the description on how to modify and extend all steps to obtain multiple subdivisions in linear time, which requires both algorithmic changes, as well as a heavily modified runtime analysis. Finally, Section 4 gives a short computational study which shows the effectiveness of this algorithm.

2 The Boyer-Myrvold Planarity Test

The test starts with a depth first search on the (not necessarily connected) input graph, which divides the edge set into DFS-forest edges and into backedges, pointing to nodes with smaller *depth first index* DFI. The aim is to construct a planar drawing based on the DFS-forest, by successively embedding all backedges in descending DFI order of their end vertices. Throughout this paper, let v be the current vertex to embed. Any backedge in the embedding step of vertex v ending on v is called *pertinent* and will be embedded, if this is possible while maintaining planarity. In the beginning, each DFS-edge is separated from its adjacent vertex with lower DFI and joined to a new *virtual* vertex. Therefore it represents a biconnected component *(bicomp)* in the beginning, which grows when backedges are embedded.

To identify involved bicomps during such an embedding, the *Walkup* is called for each start node of a pertinent backedge. A bicomp consisting of only one DFS-edge and its adjacent vertices is called *degenerated.* The Walkup marks the involved subgraph and classifies nodes as *pertinent*, *external* or inactive. A node w is called *pertinent*, if there exists a pertinent backedge $\{w, v\}$ or if w has a child bicomp in the DFS-tree which contains a pertinent node. A node w is called *external*, if there exists a backedge $\{w, u\}$ with u having a smaller DFI than v, or if w has a child bicomp containing an external node. A node that is neither pertinent nor external is called inactive. Bicomps are called pertinent or external if they contain pertinent or external vertices, respectively. The Walkup traverses a unique path from w to v on the external face of bicomps for every pertinent backedge $\{w, v\}$. We denote this path as the *backedge path* of $\{w, v\}$.

The *Walkdown* attempts to embed all pertinent backedges and merges the bicomps between their start and end vertex in the DFS-tree to a new, larger bicomp. It is invoked twice for each child bicomp of v: once in a counterclockwise direction around the external face of the child bicomp, and once in the clockwise direction. Using the classification of nodes from the Walkup, the Walkdown embeds only backedges which preserve planarity in the embedding. If any backedge cannot be embedded, the graph is not planar and a subdivision is extracted; otherwise a planar embedding is found. Since non-embeddable backedges can only occur when both Walkdowns stop on external vertices which are not pertinent, such a situation is called a *stopping configuration*. We call unembedded pertinent backedges caused by a specific stopping configuration *critical*. Let $b = \{w, v\}$ such a critical backedge. The first node in the backedge path of b which is contained in the same bicomp as both stopping vertices are, is called *critical node*. We denote the part of the backedge path from w to this critical node *critical back path*.

3 Extracting Multiple Kuratowski-Subdivisions in Linear Time

As opposed to the Boyer-Myrvold planarity test, the number of edges cannot be bounded linearly by the number of vertices. Since every algorithm has to read the input graph and to output all identified Kuratowski subdivisions, $\Omega(n+m+\sum_{K\in\mathcal{S}} |E(K)|)$ is a lower bound for the runtime and our algorithm is therefore optimal for the extracted number of Kuratowski edges.

3.1 Overview

The original planarity test terminates when a stopping configuration is found. It is possible to extract a Kuratowski subdivision for each critical backedge of this stopping configuration. To obtain more, we have to proceed with the algorithm. This bears problems, because the embedding has to be maintained planar, which is impossible if it contains Kuratowski subdivisions. The idea is to identify all critical backedges in the given stopping configuration and delete them. After that, the bicomp B containing the stopping configuration is not pertinent anymore and it is necessary to continue at the situation directly before the planarity test descended to B. This allows finding the next stopping configuration, provided that there exists any on the current embedding step of vertex v. See Algorithm 1 for an overview of these steps in the embedding process of a single vertex v.

Algorithm 1 Embedding tasks of a vertex v

```
for all pertinent backedges p ending at v do
    Walkup(p)                                                              ▷ Sect. 3.3
end for
for all DFS-children c of v do
    stop ← Walkdown(c)                                                     ▷ original Walkdown
    while stop ≠ ∅ do
        Find all critical backedges of the stopping configuration stop     ▷ Sect. 3.4
        Extract multiple subdivisions for each critical backedge           ▷ Sect. 3.4
        Delete critical backedges and update the classification of nodes  ▷ see [5]
        Find reentry_point for further embedding                           ▷ Sect. 3.2
        stop ← Walkdown(reentry_point)                                     ▷ iterated Walkdowns
    end while
end for
```

Unfortunately, almost all time-bounds given in [5] loose validity with this approach, and a new runtime analysis of this extended algorithm is necessary. The key to a linear time bound is to compensate additional costs during Walkup, Walkdown and extraction by the amount of extracted Kuratowski edges. We will first describe how to find the correct reentry point after a stopping configuration was found and removed. In Section 3.3, we discuss how to modify the Walkup, in order to allow efficient operations used in the later steps of the algorithm. Section 3.4 deals with the efficient extraction phase. Finally, the overall runtime of the extended algorithm is analyzed in Section 3.5.

Of course there are graphs with exactly one Kuratowski subdivision. Hence, we do not ensure any lower bound other than 1 for the number of extracted Kuratowski subdivisions of non-planar graphs. But in practice, the quantity is high as discussed in Section 4. Formally, our algorithm guarantees:

Lemma 1. *We find at least one unique Kuratowski subdivision for each backedge per stopping configuration.*

We also guarantee the following, which can be applied iteratively after each Kuratowski extraction:

Lemma 2. *Whenever the algorithm extracts a Kuratowski subdivision using a backedge b, and there exist additional Kuratowski subdivisions without b, we will find at least one more unique Kuratowski subdivisions.*

3.2 Finding the Reentry Point for Further Embeddings

Let v' be the virtual node of v adjacent to the DFS-child c of v from the current Walkdown. We call the bicomp which has v' as its root, the *forebear bicomp*, the others are called *non-forebear bicomps*. The Walkdown can be run unmodified, as long as no stopping configuration occurs. The same holds if a stopping configuration occurs on the forebear bicomp due to embedded pertinent backedges, since this represents the last stopping configuration in the Walkdown.

Otherwise, the Walkdown has to be modified. Let A be the non-forebear bicomp containing the stopping configuration, T the subtree of all pertinent bicomps with the bicomp containing v' as root and D the parent bicomp of A in T (cf. Figure 1). Any bicomp in T has exactly those bicomps as children which are referenced in the `PertinentRoots` lists of its nodes, as proposed in [5]. In Figure 1, the bicomp tree T consists of the (degenerated) forebear bicomp $\{v', c\}$ and the non-forebear bicomps A, B, C and D. The Walkdown stops at A, deleting the critical backedges incident to w_1 and w_2 after the extraction of all Kuratowski subdivisions induced by these backedges. Afterwards, A is not pertinent anymore and its `PertinentRoots` list entry on the parent node z_1 in D must be deleted. As there exists another item in that list, we continue the Walkdown at z_1 and find another stopping configuration in bicomp B. The general rule is that the Walkdown continues on z_1 until the `PertinentRoots` list of z_1 is empty.

At last, z_1 is not pertinent anymore. Furthermore, *short-circuit edges* from the root r of D to both external vertices in each direction (z_1 and z_2) have been embedded in the past. Note that z_1 and z_2 are the first possible parent nodes of bicomps containing stopping configurations in D; otherwise the Walkdown would have stopped before.

These short-circuit edges permit a $O(1)$-traversal to the other external vertex z_2, where the Walkdown extracts all stopping configurations of child bicomps (bicomp C in Figure 1), analogously to z_1. Finally, we check whether D itself contains a stopping configuration by extracting all remaining critical edges, since we know that D contains two stopping vertices. In our example, the backedge starting at w_5 induces a subdivision and can be deleted after the subdivision's extraction. This procedure is iterated with the next father bicomp in the DFS-tree until the forebear bicomp is reached or a pertinent backedge is embedded. In the latter case, all preceding bicomps are embedded and the Walkdown continues at the forebear bicomp.

The crucial point in this scheme is the traversal to a bicomp, where no backedge can be embedded, i.e., a bicomp that contains a stopping configuration: we then modify the embedding to what it would have been, if no critical backedges on this bicomp would have existed. Finally, the Walkdown is restarted on the very node where the previous Walkdown started to descent to this bicomp, which – after the modifications – is not necessary anymore.

3.3 Walkup

Additionally to the `PertinentRoots` list and `BackedgeFlags` of the original planarity test, we now have to collect some more information during the Walkup. For every visited node n, we store a link `LinkToRoot` to the root node of the bicomp of n. This can be done efficiently by using a stack for all visited nodes of the bicomp during the Walkup. Furthermore, a list named `PertinentNodesAfterWalkup` of all pertinent nodes of each bicomp B is created. This is stored at the root node of B by collecting the nodes during the Walkup in a list. Whenever we reach the bicomp root or a node with set `LinkToRoot`, we can add the collected vertices

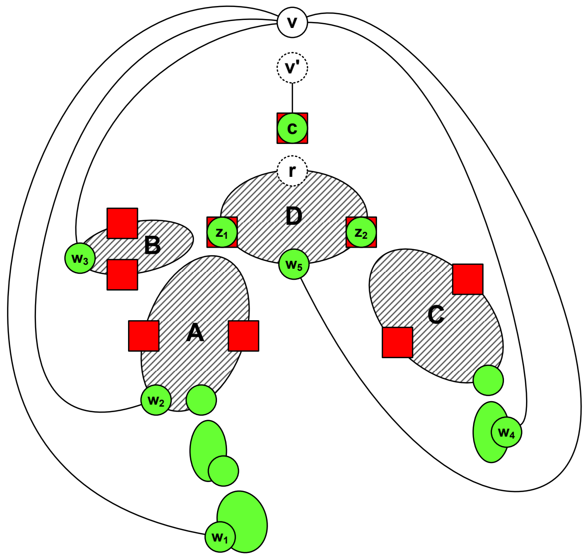


Fig. 1. An example of finding the reentry point in the situation where a stopping configuration is found by a counterclockwise Walkdown. Square nodes refer to external vertices whereas circular, light gray nodes denote pertinent vertices. Virtual vertices are depicted by a dotted line.

in $O(1)$ time to the list of the bicomp root. Once established, this list is not modified until v is completely embedded.

It is useful to be able to distinguish the backedges incident to different virtual vertices v' of v, since they will be embedded in different subtrees later on. This can be done by storing v' as the `HighestVirtualNode` for each backedge $\{w, v\}$. To obtain v' for a given backedge p, Walkup(p) marks each visited node with p. If the Walkup ends on a virtual node of v, we can store this node as the `HighestVirtualNode(`p`)`. Otherwise, Walkup(p) stopped on an already visited vertex which was traversed during the Walkup of another backedge q. Since both Walkups met, the subtrees are identical and so are the `HighestVirtualNodes` of p and q. The latter can be looked up in $O(1)$, and we hence identified `HighestVirtualNodes(`p`)`. This allows us to easily generate a list `Backedges-OnVirtualNode` for each virtual node v' of v containing the backedges belonging to the pertinent subtree with root v'.

3.4 Extraction

Overview. The extraction starts whenever the Walkdown halts on some stopping configuration in a bicomp B. We will first clarify how the critical backedges which belong to this stopping configuration can be computed in the next subsection *"Extraction of Critical Backedges"*. Each critical back path of those backedges induces one or more Kuratowski subdivisions of a specific minor-type, which has to be known prior to the subdivision's extraction.

To obtain this minor-type, two unembedded paths in the underlying DFS-tree from the stopping vertices to nodes with lower DFI than v are selected in time linearly to their lengths. As different such paths yield different Kuratowski subdivisions, we can use different ones to obtain multiple unique subdivisions. Additionally, the *highest-xy-path* of the critical node w is needed to determine the minor-type. As defined by Boyer and Myrvold, the highest-xy-path obstructs the inner face of B and consists of the external face part on the top of the former, now embedded, bicomp which contains w. This path can be computed in $O(n)$, but this would result in a superlinear overall runtime. Hence we develop a more efficient way by first extracting the more general *highest-face-path* efficiently and use this to obtain the highest-xy-paths for the critical nodes. These steps are described in the subsections *"Extraction of the Highest-Face-Path"* and *"Extraction of all Highest-XY-Paths"*. After the minor-type is determined, all remaining parts of the Kuratowski subdivision can be extracted from the DFS-tree using only external faces of the involved bicomps. This requires time linearly to their lengths. Finally, all critical backedges of the stopping configuration as well as the involved `PertinentRoots` and `BackedgeFlags` are deleted.

Extraction of Critical Backedges. Let x and y be the two stopping vertices on the bicomp B, and r the root of B. Neither x, nor y, nor any node on the external face paths $r \to x$ and $r \to y$ can be pertinent; otherwise the Walkdown would not have stopped at x and y. The critical back paths of the critical backedges end on the external face of B between x and y. We distinguish between two cases depending on the type of B.

Bicomp B is a forebear bicomp. All pertinent backedges of the current Walkdown are contained in the `BackedgesOnVirtualNode(`r`)` list. For each entry, we can check in $O(1)$ whether it is embedded. If not, the backedge is critical. This yields an overall running time of $O(n + m)$ over all embedding steps, since all critical backedges are deleted afterwards and no further stopping configuration can exist.

Bicomp B is a non-forebear bicomp. Consider the DFS-subtree T of pertinent bicomps with B as root bicomp. We start a preorder traversal through T by using the `PertinentNodesAfterWalkup` lists on the roots of all bicomps. These lists can contain nodes that are not pertinent any more due to extractions of other stopping configurations. Hence we have to check each item for pertinence; every non-pertinent entry is deleted. The remaining nodes are the critical nodes and we check their `BackedgeFlag` property. If this flag is set, the associated backedge must be critical and is therefore included in the list of critical backedges. Note that the remaining nodes, independent of their `BackedgeFlag`, may have non-empty `PertinentRoots` lists. After all critical backedges starting at the current bicomp were found, the preorder traversal iterates the process on each child bicomp given by its `PertinentRoots` lists recursively.

All tests on the nodes can be performed in constant time. The size of the tree T itself is bounded by the costs of the corresponding Walkup invocations, because at least one node was traversed for each pertinent bicomp. Moreover, a non-pertinent node in the `PertinentNodesAfterWalkup` list can only happen as a result of an earlier extracted stopping configuration. The only other reason would be that a pertinent backedge has been embedded on B, which is contradictory to the assumption. Each of the at most m stopping configurations in all embedding steps produces at most one non-pertinent entry in a `PertinentNodesAfterWalkup` list. Hence the overall runtime is bounded by the Walkup time.

Furthermore all critical nodes in B are necessary for the minor-type classification and for the extraction of Kuratowski subdivisions. We can obtain all critical nodes in B efficiently by testing the `BackedgeFlag` for each entry of the `PertinentNodesAfterWalkup` list of r.

From this description we can conclude:

Lemma 3. *The asymptotic runtime for obtaining all critical backedges of a stopping configuration is bounded by the Walkup costs.*

Extraction of the Highest-Face-Path. In order to extract all highest-xy-paths efficiently, we first require a *highest-face-path* of the bicomp B. See Figure 2 for a visualization of the following explanations. We obtain the highest-face-path by temporarily deleting all edges incident to its root r except for the two edges $s = (r, a)$ and $t = (r, b)$ on the external face (ignoring any short-circuit edges). Thereby, B breaks into multiple sub-bicomps; we also delete all *separated* sub-bicomps, i.e., the sub-bicomps which do not contain r. Consider the inner face f containing a, r, and b. The highest-face-path is the path $a \to b$ on the boundary of f not traversing r.

It is possible to extract the highest-face-path in time $O(|B|)$, if B is properly embedded. But since the planarity test performs implicit flips on bicomps, we do not know whether the adjacency lists of the nodes are in clockwise or counterclockwise order, and we would have to establish the correct orientation for each node of B first. This would require a traversal of the underlying DFS-tree and would result in a superlinear overall computation time. Hence, this approach is not suitable and instead we have to identify the highest-face-path with inconsistent node orientations.

Therefore, it is not possible to easily walk along f. The idea is to use the still existing external face links – introduced in the original planarity test – of the former, now merged bicomps in B. These *external-links* of a node referred to the two incident edges on the boundary. These edges can be compared in a traversal of the external face in order to find the correct direction to proceed, even when some nodes are not oriented correctly. To use the former external-links in a traversal inside of B, we have to analyze the general structure of B first:

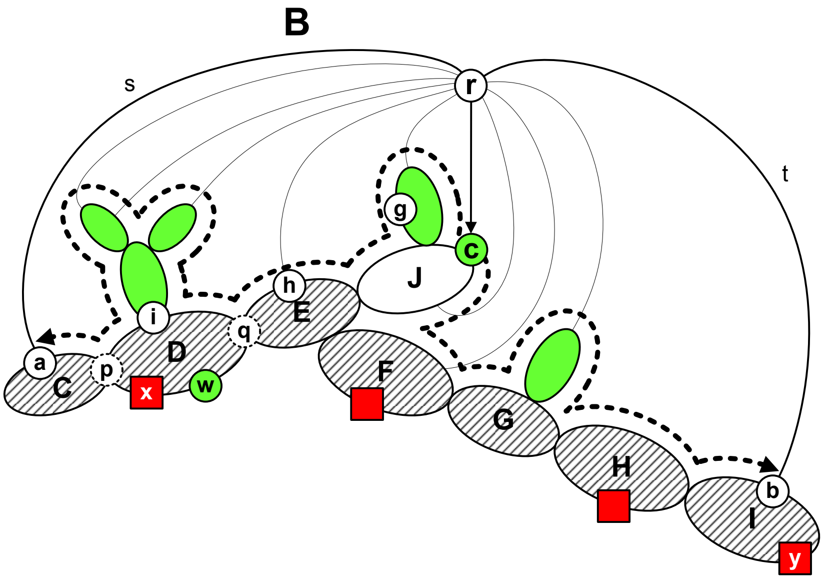


Fig. 2. The structure of the bicomp B containing former bicomps. The hatched former bicomps form the bottom chain. The extraction of the highest-face-path starts at the inner vertex c in both directions (thick dotted arrow lines) and ends on nodes a and b.

Bicomp structure. The external face of every non-degenerated forebear bicomp contains at most one embedded backedge for each of the two Walkdowns formerly started at r. It also may contain an edge connecting the root and the non-root node with least DFI. However, in all cases these edges are incident to the virtual root node. The remaining set of edges on the external face consists of the lower parts of now connected, former bicomps. We denote this sequence of former bicomps which lie on the external face the *bottom chain* of B, cf. Figure 2. A *merge node* is a node shared between two adjacent bicomps of the bottom chain (e.g. the nodes p and q in Figure 2), or one of the two end nodes a and b. Given a former bicomp U in the bottom chain, the path on the upper part of U connecting the two contained merge nodes is identical to the highest-xy-path of a critical back path ending at U. This fact is the key for the later extraction of all highest-xy-paths.

As B is not degenerated, the bottom chain exists. Let c be the unique non-virtual node of B with least DFI. Let E be the former bicomp of the bottom chain which contains the node with smallest DFI: if c is not contained in E, inner bicomps exist. The goal is to traverse these inner bicomps from c to the bicomp E on their boundaries by using the former external-links. Since the node orientation is unknown, both directions on the boundary are traversed in parallel. When both traversals reach the root node of E, these traversals continue with paths to the left and to the right on the top of the bottom chain, i.e., they identify the highest-face-path of B.

The main issue is that we somehow have to choose the proper successor bicomps. Moreover, most of the former external-links at merge nodes and end nodes of embedded backedges have been modified during the Walkdown. On merge nodes, only those former external-links remain unmodified, which refer to the external face of B.

Therefore, we store a backup copy *old-links* of the former external-links on each bicomp root traversed during the Walkup. These roots will become merge nodes during the Walkdown, and all old-links on them refer to the former external-links of the last embedded successor bicomp in the DFS-tree.

Hence, the traversal works as follows: We start with the traversal at c, checking each external-link of c for identity with s and t. The type of the traversal is determined by the number of such external links:

One external-link refers to s or t: The node c lies on the external face of B and the first Walkdown was able to embed a backedge to r. Then, c is contained in E as its root node and it is sufficient to start a single walk following the unique old-link of c, which is not identical to an external-link of c. Note that it is possible that more than one bicomp has been merged to c and that the old-links always refer to the last one.

Otherwise: None of the external-links refer to s or t and c is therefore either an inner vertex or the root of E which lies on the external face of B. The former induces inner bicomps along a non-empty path from c to the root of E. Either way, both directions on the boundary of former bicomps have to be traversed to obtain the highest-face-path of B. These traversals start on the neighboring edges of $r \to c$ in the adjacency list of c. Note that this is independent of the orientation of c.
During the traversals, E can be determined as the last bicomp, whose root node is visited by both traversals. Starting with this root, all traversed nodes are stored in two separate lists, one for each traversal direction. We obtain the highest-face-path of B by appending the reversed second list to the first one. All walks check on each visited node z whether z is identical to a or b in $O(1)$. If so, the walk is finished.

We clarify how to obtain the correct next node in all traversals and how to descent to the correct bicomps while deleting node sequences from separated bicomps, as those are not part of the highest-face-path of B. Therefore, we distinguish between three cases for a visited node z, depending on the last embedding operation performed on z by any of the two Walkdowns which started at r:

None: (See, e.g., node g in Figure 2). The external-links can be used to walk on the external face of the current former bicomp in $O(1)$.

Embedding of a backedge on z: (See, e.g., node h in Figure 2). This backedge has been embedded inside of B, since z is not an end node of the bottom chain. None of both external-links of z refer to the edge from which we came on our traversal. Instead, the embedded backedge is linked and we can use the other external-link.

Embedding of a child bicomp of z: (See, e. g., nodes i and q in Figure 2). Both external-links of z have lain on the former external face directly after the merge operation due to an embedded backedge on a successor bicomp. Since we want to traverse the inner face containing r, we have to choose the unique old-link which is not identical to an external-link.

All nodes can be classified in $O(1)$ according to these three cases. During the whole traversal, all visited nodes are saved on a stack. If a node is visited twice, this node is a merge node to an inner, separated sub-bicomp, whose boundary is not part of the highest-face-path. Then, all nodes between the two occurrences are deleted from the stack.

Note that it is possible that former bicomps are degenerated. In that case some old-links refer to the same edge. However this does not result in ambiguity.

We store the highest-face-path on the unique vertex c in B, since later extractions might need it as well. Whenever a highest-face-path has to be computed in consequence of an embedding of B within a larger bicomp B^*, B will play the role of a former bicomp. Since we only traverse the external faces of former bicomps, we will never again traverse the interior of B. Hence, and since the traversals require $O(1)$ time for each vertex, we obtain:

Lemma 4. *All highest-face-paths which occur during the algorithm can be computed and maintained in $O(n + m)$.*

Extraction of all Highest-XY-Paths. For every given critical node w between two stopping vertices of a stopping configuration, we have to compute its highest-xy-path. As before, see Figure 2 for a visualization of the following descriptions. Due to the discussed structure of bicomps, the node w is contained on the external face of a former bicomp D of the bottom chain of B. Let p and q be the two merge nodes in D. It is not clear whether the highest-xy-path of w actually exists, because, e.g., D could be degenerated. However, this can be checked in $O(1)$ by testing $w \neq p$ and $w \neq q$. Furthermore, all merge nodes on the bottom chain can be marked with increasing positive numbers for the left-hand traversal and with decreasing negative numbers for the right-hand traversal.

It would not be efficient to walk along both paths starting on w and traversing the external face of B: there are multiple Kuratowski minor-types which do not require both paths and we would therefore loose our linear runtime. Therefore, we use the shorter path by starting walks with the external-links of w in both directions in parallel, until a node marked with a number was found. It may happen that a short-circuit edge to r will be traversed due to a stopping vertex in D. In that case, the neighboring edge not being an external-link is chosen in order to stay on D.

Let p be that merge node we find first, and let it be marked with a positive number. Therefore p lies on the traversal to the left. As the node orientation is still unknown, we do not know in which direction the highest-xy-path of w starts on p in the highest-face-path. Once more, we can use the old-links: if the last traversed edge is not identical to an old-link of p, we know that p lies on the left-hand site of D. In that case, the highest-xy-path has to be the part of the highest-face-path which starts with p and ends with the node marked with an increased absolute value by one. Otherwise, the end node has to be the node with a decreased absolute value by one. The reverse holds, if the node p is marked with a negative number.

For the classification of the induced minor-type, the positions of p and q, relative to the stopping nodes on the external face of B, are important. Since we cannot afford to walk around D and check whether there are external stopping nodes, we store a special marker for each node of every path shortened by a short-circuit edge embedded at r. If p is not marked, the stopping vertex is located closer to r. Otherwise, the stopping vertex is either p or lies on the external face path $p \to w$ in D. This concludes the following lemma.

Lemma 5. *All highest-xy-paths required during the algorithm, as well as the relative position of their end nodes, can be computed in time linearly dependent on their lengths.*

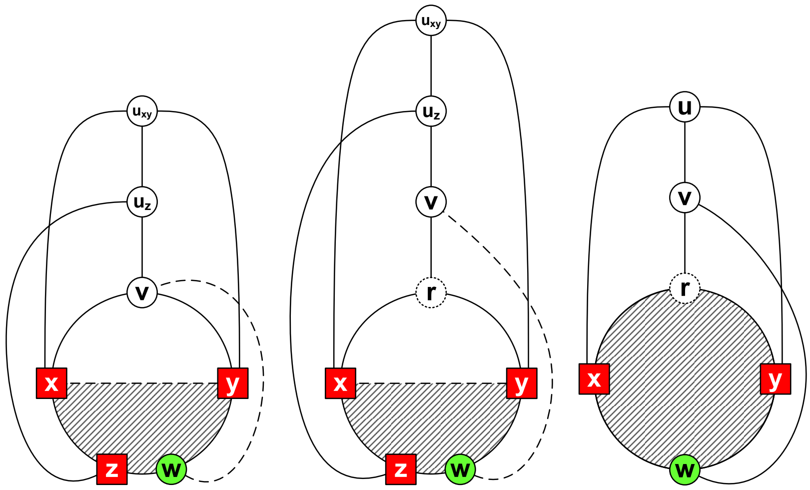
Extraction of Kuratowski Subdivisions. The prior sections dealt with the problem of efficiently obtaining multiple stopping configurations. We now address the problem to extract multiple Kuratowski subdivisions out of each single stopping configuration. Whenever a stopping configuration occurs, the appropriate critical back path for each critical backedge is computed. Along with the highest-xy-path, the minor-type of the induced Kuratowski Subdivision is obtained.

Additionally to the basic 9 minor-types by Boyer and Myrvold, we can define 7 more minor-types, by augmenting the types B, C, D and E_1–E_4 with a non-empty path $v \to r$ as in type A (cf. Figure 3). We call the resulting minor-types AB, AC, AD and AE_1–AE_4, respectively. It turns out that the Kuratowski subdivisions of these additional minor-types constitute the largest part of the extracted subdivisions in practice, see Section 4. Clearly, more than one minor-type can exist for a single critical back path.

To further increase the number of extracted Kuratowski subdivisions, we will start with focusing on the critical back paths, since nearly all minor-types needs them for constructing the subdivision. In general, such a path consists of external face parts between the roots of multiple sequential bicomps. We can therefore also extract the other parts of these external faces and combine these to obtain potentially exponentially many different critical back paths, and therefore different Kuratowski subdivisions. As a side effect, those subdivisions are all similar which can be beneficial for the application area of Branch-and-Cut algorithms.

The same technique can be used to obtain multiple external backedges, their backedge paths, and multiple paths starting at so-called *external z-nodes* (cf. [5]) in the minor-types E_1–E_5 and AE_1–AE_4.

All extracted Kuratowski subdivisions of a stopping configuration are unique. This holds for subdivisions of different stopping configurations as well, except for the minor-types E_2 and AE_2, which do not include the critical back path



(a) Minor-type E_2 (b) Minor-type AE_2 (c) Minor-type A

Fig. 3. Minor-types causing ambiguous Kuratowski subdivisions.

and thus might be extracted as minor-type A later on (see Figure 3). This can be avoided by a special marker on the external backedges, to prohibit its classification as a future critical backedge in A.

Bundle Variant. Moreover, we can extend our algorithm by a *bundle variant* in which all root-to-root paths of each involved bicomp on a critical back path are extracted. This approach increases the number of identified subdivisions dramatically, albeit on the cost of the running time. To speed up the backtracking subroutine, it is possible to use algorithms for dynamic connectivity for planar graphs [8]. This increases the overall runtime only by a factor of $log(n)$ in comparison to the linear time approach in terms of output complexity.

3.5 Runtime Analysis

As described in Section 3.2, our strategy to identify the next reentry point needs $O(1)$ time for each traversed pertinent bicomp in the bicomp-tree T. Hence, its overall running time is bounded by the Walkup costs. The Walkdown is not modified except for finding this next reentry point. Additionally, all steps to find stopping configurations and classify Kuratowski subdivisions can either be bounded by the Walkup costs as well (e.g., extraction of critical backedges, see Lemma 3) or can be done in linear total time (e.g., extraction of highest-face-paths and highest-xy-paths, see Lemmata 4 and 5, respectively).

It remains to show that the modified Walkup can be bound by a linear total of $O(n + m + \sum_{K \in \mathcal{S}} |E(K)|)$. We will only give a brief sketch of the proof, and omit a number of rather technical case differentiations. For a detailed analysis see [16].

For the minor-types E_1–E_5 and AE_1–AE_4 it is necessary to compute at least one external node strictly below the existing highest-xy-path in the stopping configuration. Such nodes are called *external z-nodes*. They are computed by checking the whole external face of the former bicomp containing the start node of the chosen critical backedge on external nodes. Since this is expensive, we restrict this computation to situations, where it is known that a minor-type E or AE must exist or the whole external face can be checked without losing linear time due to previous extracted subdivisions.

It is sufficient to consider only the costs of the Walkup, which cannot be compensated by new embedded faces or new short-circuit edges. Therefore, we only consider Walkup costs on critical backedge paths. If these are part of stopping configurations on non-forebear bicomps, the sum of all critical backedge-path costs on all forebear bicomps can be estimated as follows: we spend at most $O(n+m+\sum_{K\in\mathcal{S}}|E(K)|)$ time on the external face, and at most $O(m)$ time on inner faces containing the forebear root. Moreover, all other costs caused by stopping configurations in non-forebear bicomps are compensated by the inevitably induced minor A, since this minor-type contains all other traversed edges.

Otherwise, the stopping configuration is contained in a forebear bicomp. Since most minor-types do not contain the whole external face in their Kuratowski subdivisions, all not yet compensated costs arise on its external face. The only exception to this rule are the critical paths on minors E_2 and AE_2, which can be bound by a linear total as well. These remaining costs can be compensated by the extracted Kuratowski paths of the different minor-types. Hence we yield Theorem 1, which is optimal in terms of output complexity.

Theorem 1. *The overall running time of the algorithm is* $O(n{+}m{+}\sum_{K\in\mathcal{S}}|E(K)|)$ *and therefore linear.*

Based on this, we obtain the following related result for the computation time of the bundle variant.

Theorem 2. *The overall running time of the bundle variant of the algorithm is* $O(n+m+\log n\sum_{K\in\mathcal{S}}|E(K)|)$.

4 Experimental Analysis

We implemented the algorithm and its bundle variant as part of the open-source C++-based *Open Graph Drawing Framework* (OGDF) [1]. All tests were performed on an Intel Core2Duo E6300 with 1.86 GHz and 2GB RAM using the GNU-compiler gcc-3.4.4 (-o1).

Due to the algorithmic complexities, we simplified the steps to compute the critical backedges and highest-xy-paths. We correctly orient B in time $O(|B|)$. Although this simplification breaks the provable linear runtime, our experiments show that it does not influence the running time negatively in practice, since the

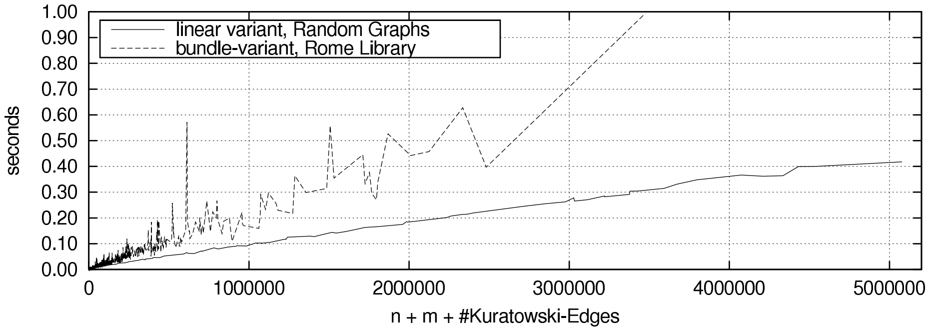


Fig. 4. Running times of the bundle variant on the Rome Library and the linear variant on the random graphs. The linear variant for the Rome Library would be nearly invisible in the very left corner of the figure. Note that the bundle variant is not linear in general.

number of extracted Kuratowski edges becomes the dominant term. The bundle variant uses a traditional back-tracking scheme and therefore does not guarantee the theoretical logarithmic bound.

We use the graphs of the well-known Rome Library [2], which contains 11528 real-world graphs with 10 to 100 nodes, 8249 of which are non-planar graphs. We also use random graphs ($n = 10 \dots 500$, $m = 2n$) generated by OGDF. Thereby we start with an empty graph on n vertices and iteratively add an edge with random start and end node, until m unique edges are added.

For the relatively small and sparse graphs of the Rome Library, the algorithm is nearly as fast as the plain planarity test itself. Each Rome graph is processed in less than 11 ms (on average: 1.3 ms). The average amount of extracted Kuratowski subdivisions per 100-node graph is 255, containing in total 12214 Kuratowski edges. It is interesting that the average size of the subdivisions grows approximately with $n/2$ throughout all tests.

More Kuratowski subdivisions are obtained by the bundle variant. Thereby, each graph is processed in less than 1 sec (but on average less than 7 ms), extracting up to 3.5 million Kuratowski edges at some graphs (see Figures 4 and 5). There are 2912 subdivisions on average per 100-node graph with 136027 Kuratowski edges.

On the random graphs, the number of identified Kuratowski subdivisions increases dramatically for the bundle variant, such that a full computation becomes prohibitive. In practice, one can of course stop the computation after a certain amount of Kuratowski subdivisions has been identified.

Hence, we restrict our test to the linear variant for these random graphs (see Figures 4 and 5). Each graph needs less than 430 ms (126 ms on average), extracting up to 25000 Kuratowski subdivisions per graph containing 5 million Kuratowski edges. The average number of Kuratowski subdivisions is 8813 per graph with 1.3 million Kuratowski edges.

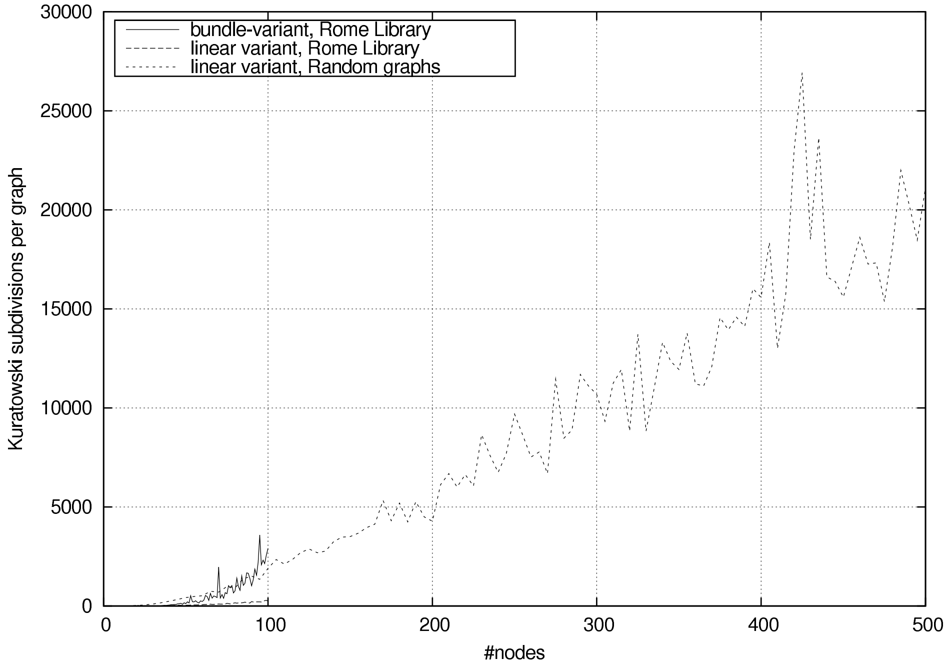


Fig. 5. Number of Kuratowski subdivisions of each test.

Overall, the experiments show a linear running time, despite the aforementioned simplifications of the algorithm. The minor-types are dominated by the types AE_1–AE_4, which constitute 60%–90% of all subdivisions on graphs with at least 100 nodes.

References

1. OGDF - Open Graph Drawing Framework. University of Dortmund, Chair of Algorithm Engineering, 2007. Website under Construction.
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(5-6):303–325, 1997.
3. J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In G. Liotta, editor, *Proceedings of the 11th International Conference on Graph Drawing*, volume 2912 of *LNCS*, pages 25–36. Springer-Verlag, September 2003.
4. J. M. Boyer and W. J. Myrvold. Stop minding your P's and Q's: A simplified O(n) planar embedding algorithm. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 140–146, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.
5. J. M. Boyer and W. J. Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. *Journal of Graph Algorithms an Applications*, 8(3):241–273, 2004.

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. *Discrete Optimization*, 2007. to appear.
7. C. Buchheim, D. Ebner, M. Jünger, G. W. Klau, P. Mutzel, and R Weiskircher. Exact crossing minimization. In P. Healy and N. S. Nikolov, editors, *Proc. GD '05*, volume 3843 of *LNCS*, pages 37–48. Springer-Verlag, 2006.
8. D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. R. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. *J. Algorithms*, 13(1):33–54, March 1992. Special issue for 1st SODA.
9. H. de Fraysseix and P. Ossona de Mendez. On cotree-critical and dfs cotree-critical graphs. *Journal of Graph Algorithms an Applications*, 7(4):411–427, 2003.
10. H. de Fraysseix, P. Ossona de Mendez, and P. Rosenstiehl. Trémaux Trees and planarity. *Int. J. Found. Comput. Sci.*, 17(5):1017–1030, 2006.
11. H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by Trémaux orders. *Combinatorica*, 5(2):127–135, 1985.
12. J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.
13. M. Jünger and P. Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica, Special Issue on Graph Drawing*, 16(1):33–59, 1996.
14. A. Karabeg. Classification and detection of obstructions to planarity. *Linear and Multilinear Algebra*, 26:15–38, 1990.
15. K. Kuratowski. Sur le problème des corbes gauches en topologie. *Fundamenta Mathematicæ*, 15:271–283, 1930.
16. J. M. Schmidt. Effiziente Extraktion von Kuratowski-Teilgraphen. Diploma thesis, Department of Computer Science, University of Dortmund, March 2007.
17. S. G. Williamson. Depth-first search and Kuratowski Subgraphs. *J. ACM*, 31(4):681–693, 1984.