

Simplified $O(n)$ Planarity Algorithms

John M. Boyer
PureEdge Solutions Inc.
jboyer@PureEdge.com; jboyer@acm.org

Wendy Myrvold *
University of Victoria
wendym@csr.UVic.ca

December 3, 2001

Abstract

A graph is *planar* if it can be drawn on the plane with vertices at unique locations and no edge intersections except at the vertex endpoints. Due to the wealth of interest from the computer science community, there are a number of remarkable but complex $O(n)$ planar embedding algorithms. This paper presents an $O(n)$ planar embedding algorithm that avoids a number of the complexities of prior approaches (an early version of this work was presented at the January 1999 Symposium on Discrete Algorithms). In July 1999, Shih and Hsu published a new planarity algorithm based on a data structure they call a PC-tree, which is a simplification of a PQ-tree that utilizes some results also used in our planarity algorithms. This paper also discusses some errors in the PC-tree formulation that prevent it from being correct and $O(n)$ as published. Finally, our new formulation is easy to prove correct and $O(n)$, and it extends well to related problems. This paper also presents a simplified $O(n)$ Kuratowski subgraph isolator, and further extensions will be presented in future papers (a number of which can be found currently in Boyer's dissertation).

1 Introduction

An *undirected graph* G contains a set V of n vertices and a set E of m edges, each of which corresponds to an unordered pair of vertices. In an undirected graph, the vertices associated with an edge are called the *endpoints* of the edge, and an edge is *incident* to its endpoints. An edge with endpoints u and v is denoted (u, v) . A *loop* is an edge of the form (u, u) , and a *multiple edge* is an edge that occurs more than once in E (if there are multiple edges, then E is not a set but rather a multiset). A *multigraph* is a graph that permits loops and multiple edges (some texts forbid loops [8]), and a *simple graph* is a graph that forbids loops and multiple edges. In this paper, graphs are undirected and simple unless stated otherwise.

In a graph G , vertex u is *adjacent* to vertex v , or equivalently v is a *neighbor* of u , if (u, v) is an edge in $E(G)$. The subset of vertices adjacent to a vertex u is called the *neighborhood* of u . The *degree* of a vertex u is the number of non-loop edges containing u as an endpoint plus twice the number of loops of the form (u, u) . In a simple, undirected graph, the degree of a vertex u is equal to the size of its neighborhood, and each neighbor v of a vertex u is also adjacent to u . A *walk of length k* is a sequence $P = (v_0, e_0, v_1, e_1, v_2, \dots, e_{k-1}, v_k)$ of

*Supported by NSERC.

alternating vertices and edges from a graph G , with $e_i = (v_{i-1}, v_i)$ an edge of G for i from 1 to k . A *cycle* is a walk of length greater than two with no repeated vertices except $v_0 = v_k$. A *path* is a walk with no repeated vertex.

A graph is often drawn using points for the vertices and lines (possibly curved) for the edges. A *planar representation* is a drawing of a graph on a plane such that the vertices are placed in distinct positions and no two edges intersect except at common vertex endpoints. A planar representation of a graph divides the plane into connected regions, called *faces*, each bounded by edges of the graph [8, p. 68]. A region of finite area is called a *proper face* and is bound by a cycle. The *external face* is the plane less the union of the proper faces and the points occupied by the planar representation of the graph. In general, the boundary of the external face is a walk, and the term external face is often used to refer to the vertices and edges along the bounding walk. Figure 1(a) shows an example graph with four vertices and six edges. Figure 1(b) shows a planar representation of the same graph.

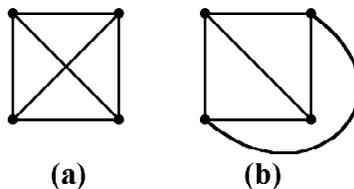


Figure 1: An Example Graph and a Planar Representation of the Graph

A graph is *planar* if it is possible to create a planar representation of the graph, and a *non-planar* graph is a graph for which there is no planar representation. A *planarity testing algorithm* determines if a graph has a planar representation. A *planar embedding algorithm* not only tests planarity but also indicates the clockwise order of the neighbors of each vertex of a planar graph. Generating the specific vertex positions and edge shapes in a planar representation is often viewed as a separate problem, in part because it is application-dependent. For example, our notion of what constitutes a suitable rendering of a graph may differ substantially if the graph represents an electronic circuit versus a hypertext book. Hence, a data structure in which the representation of each vertex contains a clockwise-ordered list of its neighbors is called a *combinatorial planar embedding* and is considered to be a simple certificate of planarity that contains sufficient information for subsequent graph drawing algorithms and many other planar graph algorithms.

While the combinatorial planar embedding provides a simple certificate of planarity, the first characterization of planarity by Kuratowski [13] shows that it is also possible to create a simple certificate of non-planarity for non-planar graphs. A *subgraph* of a graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. A graph G is *isomorphic* to a graph H if there exists a bijection $f : V(G) \rightarrow V(H)$ such that $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(H)$. An *edge subdivision* replaces an edge (u, v) with a degree two vertex w plus the edges (u, w) and (w, v) . The inverse operation, a *series reduction*, replaces a degree two vertex w and its incident edges, (u, w) and (w, v) , with a single edge (u, v) . A graph G is *homeomorphic* to a graph H if G can be made isomorphic to H by applying zero or more subdivisions and series reductions. For graphs G and H , we say that G is an *H homeomorph* if G is homeomorphic to H . Kuratowski proved that a graph is planar if and only if it contains no subgraph homeomorphic to either of two graphs, which are denoted K_5 or $K_{3,3}$ and depicted in Figure 2.

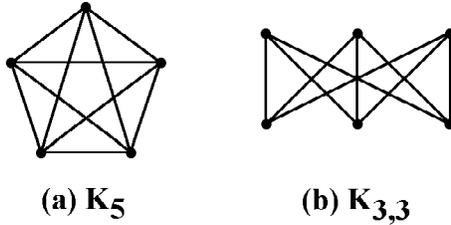


Figure 2: The Planar Obstructions K_5 and $K_{3,3}$

In some applications, finding a Kuratowski subgraph is a first step in eliminating problem areas in the graph. For example, in a graph representing an integrated circuit, an edge intersection would be indicative of a short-circuit, which could be repaired by replacing the crossed edge in an identified Kuratowski subgraph with a subcircuit of exclusive-or gates [15]. Due to Kuratowski's theorem, a non-planar graph must contain a subgraph homeomorphic to K_5 or $K_{3,3}$, and a Kuratowski subgraph isolator must return a minimal non-planar subgraph containing only five vertices of degree four or six vertices of degree three, called *image vertices*, plus distinct paths containing zero or more degree two vertices that connect the image vertices such that the resulting graph is a K_5 or $K_{3,3}$ homeomorph.

A graph is *connected* if, for every pair of vertices u and v , there exists a path (u, \dots, v) . A *connected component* of a graph is a maximal connected subgraph. Vertex v is a *cut vertex* of graph G if the removal of v and its incident edges increases the number of connected components in the resulting graph. For example, the connected graph in Figure 3(a) contains a cut vertex v whose removal, along with its incident edges, separates the graph into two connected components as shown in Figure 3(b). A graph with no cut vertices is *biconnected*. A *biconnected component* of a graph is a maximal biconnected subgraph. If a graph is not biconnected, the biconnected components are said to be separable by the cut vertices in the graph. For example, the graph in Figure 3(a) has two biconnected components as shown in Figure 3(c). Note that the cut vertex v is considered to be part of each biconnected component that contains it. As such, the bounding walk for the external face of a biconnected component is a cycle (except of course for the special case of a biconnected component containing only a single edge).

Linear time algorithms for identification of cut vertices and biconnected components using the well-known graph processing method of depth first search were first discussed by Tarjan [19]. Depth first search (DFS) on graphs operates in the same way as the well-known pre-order tree traversal method, except that DFS on graphs must terminate traversal on edges that lead back to previously visited vertices. Edges that lead to new vertices are called *tree edges*, and edges that lead to previously visited vertices are called *back edges*. The tree edges collectively form a spanning *Depth First Search Tree* in each connected component of a graph. Each vertex v is assigned a number, the *depth first index* or DFI, which indicates how many vertices were visited by the depth first search method prior to visiting v . The *root* of a DFS tree is the first vertex visited in a connected component, so it has the least DFI in the connected component. An *ancestor* of a vertex v is any vertex on the path of tree edges from v to the root, excluding v . A *descendant* of a vertex v is any vertex for which v is an ancestor. The endpoints of a back edge share the ancestor-descendant relationship. The *parent* of a vertex v is the ancestor of v adjacent to v by a tree edge. A *child* of a vertex v is any vertex for which v is the parent. A *subtree* is a subgraph of a tree in which a vertex v ,

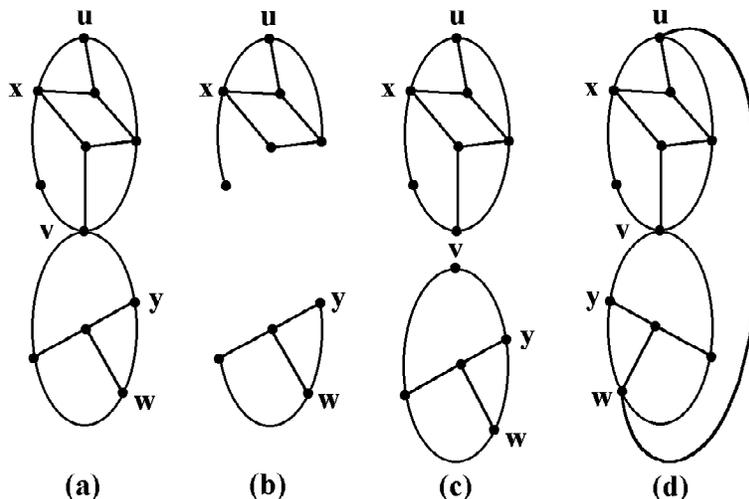


Figure 3: (a) A Cut Vertex v , (b) Removing v results in more connected components, (c) The biconnected components separable by v , (d) Adding edge (u, w) while keeping x and y on the external face bounding cycle (v is no longer a cut vertex)

called the *subtree root*, is a common ancestor to all other vertices in the subtree and which contains all tree edges having both endpoints in the subtree.

Based on a depth first search tree, we can examine more closely an interesting scenario involving the addition of an edge with endpoints u and w to the graph in Figure 3(a). Suppose u is a DFS ancestor of v , and v is an ancestor of w . Further, suppose that the graph in Figure 3(a) is a subgraph of a larger graph G that contains a DFS ancestor t of the vertex u . Firstly, since the removal of v would no longer increase the number of connected components, v would no longer be a cut vertex. This example demonstrates the fundamental operation performed by our new algorithm, which is to embed one edge e at a time and merge any biconnected components that are no longer separable by a cut vertex when e is added. Figure 3(d) depicts how our algorithm would perform this fundamental operation under the assumption that x and y are endpoints of back edges from G that are incident to t . Our algorithm embeds all DFS tree edges first, then it embeds the back edges ensuring that all back edges with an ancestor endpoint of u are embedded before back edges with an ancestor endpoint that is an ancestor of u . Thus, the back edges (t, x) and (t, y) will not be embedded at the time that edge (u, w) is embedded. Our algorithm keeps vertices such as x and y on the external face bounding cycles of biconnected components. In turn, adherence to this constraint necessitates the ability to ‘flip’ the embedding of a biconnected component before merging it with another biconnected component. As shown in Figure 3(d), the biconnected component containing w is flipped before the merging the two copies of cut vertex v and adding the edge (u, w) .

Although Kuratowski was the first to characterize planarity, a theorem by Wagner [20] is more useful for the proof of correctness of this new approach to planarity testing. An *edge contraction* of an edge $e = (u, v)$ replaces u and v and their incident edges with a single vertex w whose incident edges are all edges that were incident to either u or v (except e). An edge contraction can result in multiple edges incident to w (e.g., the edges (u, x) and (v, x) are changed to be two instances of the edge (w, x)). A graph G contains a graph H

as a *minor* if a graph isomorphic to H can be created from a subgraph of G by applying zero or more edge contractions. Wagner’s characterization of planarity states that a graph is planar if and only if it contains neither K_5 nor $K_{3,3}$ as a minor.

The remainder of this paper is divided into the following sections. Section 2 reviews selected prior works. Section 3 presents the essential definitions and operations of our planar embedder and proves its correctness by demonstrating that the algorithm creates a planar embedding over the edges it embeds, and that failure to embed an edge indicates that the input graph contains $K_{3,3}$ or K_5 as a minor. Section 5 presents additional information about our data structures and their initialization and maintenance to support an efficient implementation of our algorithms. Section 6 provides a more detailed version of our embedding algorithm as well as a proof that $O(n)$ performance is achieved. Section 7 presents our new Kuratowski subgraph isolator. Section 8 presents some errors that have been found in the PC-tree algorithm. Finally, Section 9 presents concluding remarks, including some comments about the brevity of implementation of our $O(n)$ planarity algorithms.

2 Review of Selected Prior Works

The first $O(n)$ planarity test algorithm is due to Hopcroft and Tarjan [9]. The method first embeds a cycle C of the graph, then it breaks the remainder of the graph into a sequence of paths that can be added either to the inside or outside of the starting cycle. Some corrections to the algorithm appear in [6], and significant additional details are presented by Williamson [22, 24] as well as the text by Reingold, Nievergelt and Deo [17].

Despite these resources, it is difficult to envision how the tester achieves linear time on certain key parts, such as reversing a prior decision about whether to embed certain sets of paths on the inside or outside of the cycle C . Indeed, Hopcroft and Tarjan comment that the challenging part of the algorithm is the creation of good data structures to efficiently implement this method. Moreover, while their conclusion briefly sketches a method for augmenting the tester to create a planar embedder, over a decade later, Chiba, Nishizeki, Abe, and Ozawa comment that modifying the Hopcroft and Tarjan algorithm to yield a planar representation “looks to be fairly complicated; in particular, it is quite difficult to implement a part of the algorithm for embedding an intractable path” [4, p. 55].

The second method of planarity testing proven to achieve linear time began with an $O(n^2)$ algorithm due to Lempel, Even and Cederbaum [14]. The algorithm begins by creating an s, t -numbering for a biconnected input graph. One property of an s, t -numbering is that there is a path of higher numbered vertices leading from every vertex to the vertex t , which has the highest number. Thus, there must exist an embedding \tilde{G}_k of the first k vertices such that the remaining vertices ($k + 1$ to t) can be embedded in a single face of \tilde{G}_k . This planarity testing algorithm was optimized to linear time by a pair of contributions. Even and Tarjan [7] optimized s, t -numbering to linear time, while Booth and Lueker [1] developed the PQ-tree data structure, which allows the planarity test to efficiently maintain information about the portions of the graph that can be permuted or flipped before and after embedding each vertex. Chiba, Nishizeki, Abe and Ozawa [4] augmented the PQ-tree operations so that a planar embedding is computed as the operations are performed.

Achieving linear time with the vertex addition method is also quite complex [11], in part because Booth and Lueker do not include the complete set of optimized templates required to update the PQ-tree quickly [1, p. 362] but leave them for the reader to derive. There

are non-trivial rules for restricting processing to only the pertinent portion of the PQ-tree, more rules to prune the tree, then still more details to increase the efficiency of selecting and applying templates since more than one is often applied to process each vertex.

While the two major algorithms appear to approach the planarity problem quite differently, Canfield and Williamson [3] have shown that a few modifications can be made to synchronize the behavior of the two algorithms for each vertex. Williamson [23] was also the first to create an $O(n)$ Kuratowski subgraph isolator based on identifying the non-planarity conditions that arise in the Hopcroft and Tarjan method. Karabeg [12] developed a linear time Kuratowski subgraph isolator by exploiting non-planarity conditions that arise in PQ-trees.

A planarity characterization by de Fraysseix and Rosenstiehl [5] could lead to $O(n)$ planarity algorithms, though the paper contains no development of a linear time methodology. However, the characterization is important because planarity is explained in terms of conflicts between back edges as seen from a bottom-up view of the depth first search tree.

A preliminary report containing some of the results of this dissertation appeared in [2]. Our work is based on this bottom-up view, though we derived it independently of [5] as the result of trying to eliminate PQ-trees and s, t -numbering while retaining the aforementioned property of s, t -numbering that all paths lead to the final vertex. Using a bottom-up view of the DFS tree, the final vertex is the DFS tree root, which suggests a bottom-up vertex processing order. While [2] contained most of the information necessary to embed the back edges between each vertex and its depth first search descendants and to recover a combinatorial planar embedding in linear time, as an extended abstract it does not contain complete information. Specifically, it lacks details that define when the root of a biconnected component should be kept on the external face, and it does not provide the details on how to maintain and use path information for each biconnected component encountered as the algorithm works bottom-up from descendants adjacent by a back edge to the vertex being processed. This paper improves on the prior work by presenting a simpler formulation that does not even need to maintain this path information nor any external face information for biconnected component roots.

More recently, a new planarity test was presented by Shih and Hsu [18]. They develop a simplification of a PQ-tree that is critically dependent on a number of the same results that appeared earlier in Boyer and Myrvold [2]. However, since their date of submission was significantly before the publication of [2], there is clearly a case of simultaneous independent discovery of somewhat similar algorithms. However, the results of Shih and Hsu stated in [18] contain several errors, which will be discussed in Section 8.

3 New Planar Embedder

3.1 Terminology and Top-Level Algorithm

In Section 1, we introduced the fundamental operation performed by our new planarity algorithm, which is the addition of an edge that may biconnect previously separable biconnected components. Our embedding data structure, denoted \tilde{G} , is designed to maintain a collection of combinatorial planar embeddings of the biconnected components that develop as each edge of the input graph G is added to \tilde{G} . Although an $O(n)$ implementation of our algorithm can be created by caching a few extra pieces of information for each vertex and edge,

in this section we focus on a minimal representation for \tilde{G} . As mentioned in Section 1, a representation for a cut vertex must appear in each biconnected component containing the cut vertex. Other than maintaining additional adjacency lists for extra copies of cut vertices, a standard adjacency list format is sufficient for representing \tilde{G} if optimal performance is not required. Thus, the algorithm development and proof of correctness can proceed in a standard graph theoretic context with implementation details to follow in Sections 5 and 6.

The new embedding algorithm begins by creating a depth first search tree for the input graph G . Observe that a cut vertex appears in every path between its DFS children and its ancestors, except for cut vertices that are DFS tree roots, which have no ancestors. Whether or not a cut vertex is a DFS tree root, a cut vertex is the first vertex visited by depth first search in a biconnected component containing the cut vertex and a DFS child of the cut vertex. Moreover, a biconnected component B with depth first search entry point v cannot contain more than one DFS child c of v . The assumption of a second DFS child c_2 of v in B contradicts the biconnectedness of B since the depth first search finds no path from c to c_2 except through v . For these reasons, following definitions specify the nature of the extra copies of vertices that must be maintained in order to represent each cut vertex in each biconnected component containing it. A *virtual vertex* is the vertex with the least depth first index in a biconnected component. A virtual vertex representing vertex v in a biconnected component B is denoted v' , or v^c if it is necessary to identify the DFS child c of v in B . The virtual vertex in a biconnected component is the *root* of the biconnected component. A *root edge* is a DFS tree edge incident to a virtual vertex (and the DFS child of the virtual vertex). A root edge is denoted (v^c, c) or simply (v', c) since the child c is known. A *child biconnected component* of a vertex v is a biconnected component in \tilde{G} that contains the virtual vertex v^c for some DFS child c of v .

Once the depth first search of G has been performed (along with a few other preprocessing steps that cache information useful in achieving $O(n)$ performance), the algorithm embeds in \tilde{G} a root edge corresponding to each tree edge of G . Each root edge is a singleton biconnected component consisting of a vertex c and a virtual vertex representing the DFS parent of c .

The final step of the top-level algorithm, pseudo-code for which appears in Figure 4, is to embed the back edges from each vertex to its descendants, flipping and merging biconnected components as necessary. The number of edges is restricted to $3n - 5$ since planar graphs have no more than $3n - 6$ edges (we allow one extra edge so that a Kuratowski subgraph can be found if the input graph is not planar; a higher edge limit can be imposed at implementer discretion). The rationale for processing the vertices in reverse DFI order is simply that, for any step v , a partial embedding can be created in which the remaining unprocessed vertices can be embedded in the external face because each has a path of DFS tree edges leading to the DFS tree root.

The order in which back edges are embedded and the details of the biconnected component flip and merge operations are selected such that all vertices with unembedded back edges to v or its ancestors remain on the external faces of biconnected components in \tilde{G} . The following definitions support the ability to make appropriate decisions about these operations. Given a vertex x in \tilde{G} that is a descendant of the current vertex v being processed by the main algorithm loop, x is *externally active* if the input graph G contains a back edge (u, x) where u is an ancestor of v , or if x has a child biconnected component that contains at least one externally active vertex. An *externally active biconnected component* in \tilde{G} is a biconnected component that contains at least one externally active vertex. A biconnected

Figure 4: High-Level Outline of New Planarity Algorithm

- (1) Receive graph G with $n > 2$ vertices and $m \leq 3n - 5$ edges
- (2) Perform depth first search on G and other preprocessing
- (3) Based on G , create and initialize embedding \tilde{G}
- (4) Add each DFS tree edge of G to \tilde{G} as a singleton biconnected component
- (5) For each vertex v of G in reverse DFI order
- (6) Embed in \tilde{G} each back edge in G from v to a DFS descendant of v . For each such back edge (v, w) , embed (v, w) such that:
 - a) all biconnected components are merged together that will no longer be separable when (v, w) is added
 - b) any vertex x with unembedded back edges to v or DFS ancestors of v is kept on the external face (along with cut vertices separating x from v).

If embedding (v, w) requires violation of 6b,
break the loop

- (7) If one or more back edges were not embedded,
Isolate a Kuratowski subgraph

component in \tilde{G} with root r^c is *pertinent* at step v if the input graph G contains at least one back edge (v, w) not embedded in \tilde{G} , where w is in the DFS subtree rooted by c . A vertex w in \tilde{G} is *pertinent* in step v if the input graph G contains a back edge (v, w) not embedded in \tilde{G} or if w has a pertinent child biconnected component. A vertex w in \tilde{G} is *internally active* if it is pertinent but not externally active. An *internally active biconnected component* is a biconnected component in \tilde{G} that contains one or more internally active vertices and no externally active vertices. A vertex in \tilde{G} is *inactive* if it is neither externally nor internally active.

There are a couple of notable aspects of these definitions. First, they apply only to vertices, not virtual vertices. Second, since the definition of a child biconnected component states that it must be rooted by a virtual vertex, the activity or pertinence of a vertex w is affected only by unembedded back edges to the ancestors of w either directly from w or from only those vertices in DFS subtrees rooted by children of w that are not in the same biconnected component as w .

In Figure 4, the main algorithm loop body is not specified in Line 6. Rather, we have so far only said what must be done but not how. The obvious first step is to set up the pertinence and activity conditions defined above, but without the requirement of linear time performance, it is easy to create inefficient but trivially simple implementations for these

definitions (and efficient methods are presented in Section 5).

To process the back edges from v to its descendants, our algorithm performs a routine called ‘Walkdown’ on each biconnected component rooted by a virtual vertex v' . Since no back edges to ancestors of v have been embedded, each such biconnected component is a single root edge when the Walkdown is first invoked, but each one that is pertinent becomes larger as the Walkdown embeds back edges.

3.2 The Walkdown Routine

An invocation of the Walkdown routine on a singleton biconnected component B with root edge (v', c) embeds all back edges between v and the descendants of c except when a non-planarity condition is discovered, the details of which are covered in the proof of correctness in Section 4. Each such back edge to a descendant of c is embedded along the external face of B incident to the descendant and the virtual vertex $v' = v^c$. Child biconnected components of c and its descendants are merged into B as necessary such that both endpoints of each embedded back edge are in B .

The Walkdown procedure begins with a counterclockwise traversal of the external face of B starting at v' . Since B contains a single edge, there appears at first to be no difference between counterclockwise versus clockwise traversal, but the distinction becomes clear almost immediately when the root of a child biconnected component is merged with c . In the counterclockwise traversal, pertinent biconnected components are merged as necessary during the traversal in order to embed the back edges. These actions attach more vertices and edges to B such that its external face consists of more than the original root edge. If the first traversal processes all vertices on the external face of B , returning to v' , then the Walkdown terminates. However, the first traversal also stops if it encounters a *stopping vertex*, which is a non-pertinent externally active vertex. In this case, the Walkdown performs a second traversal of the external face of B starting at v' and proceeding clockwise. Again, the traversal proceeds to embed back edges and merge pertinent biconnected components until a stopping vertex is again encountered.

When a Walkdown traversal visits a vertex w , it performs two tasks. The first task is to determine whether a back edge to w must be embedded based on whether the back edge (v, w) exists in G and not in \tilde{G} . If so, then any child biconnected component roots between v' and w (obtained in the manner described below) are merged and the new back edge (v', w) is embedded such that the external face paths traversed from v' to w form a proper face with the new back edge. The second task performed when visiting w is to determine whether w has any pertinent child biconnected components. If not, then w is inactive, so the Walkdown obtains the successor s of w along the external face. On the other hand, if w has a pertinent child biconnected component, then the Walkdown obtains the root w' of a pertinent child biconnected component with preference for the root of an internally active child biconnected component, if any.

Traversal continues from w to the selected child biconnected component root w' , and a direction to continue traversal from w' must be chosen. Both external face paths originating from w' are scanned to find the first vertices x and y in both directions that are not inactive. The path to an internally active vertex is selected. If both x and y are internally active, then the choice is arbitrary. If both x and y are externally active, then the path to a pertinent vertex is selected. If both are pertinent, then the choice is arbitrary. If neither are pertinent

(i.e. if x and y are stopping vertices), then the choice is also arbitrary because a non-planarity condition discussed in Section 4 has been discovered and will soon cause the termination of the Walkdown.

Once a new non-virtual vertex in a child biconnected component has been selected for visitation according to the rules above, then w and w' as well as indicators of the direction by which w was entered and w' was exited are pushed onto a *merge queue*. The processing of the next vertex then begins. The merge queue will continue to grow until the next vertex directly adjacent to v by an unembedded back edge is encountered, at which point the merge queue will be processed so that all virtual vertices are merged with their non-virtual counterparts prior to embedding the new back edge.

Perhaps the simplest way to communicate the result of merge queue processing is to consider how its contents would be processed if it were a stack, i.e. consider processing it from the last element to the first. Each virtual vertex w' and corresponding non-virtual vertex w are popped from the stack, along with the directional information. Immediately prior to merging w' and w , w' is the root of either a biconnected component or of a connected component formed by prior merge operations. In either case, the orientations of all vertices in the component rooted by w' must be inverted, essentially flipping the component, if the direction of entry into w opposes the direction of exit from w' .

Processing the merge queue as a stack can obviously be very costly. Although we defer remarks on linear time performance until Section 5, we can give a hint of the optimization now while also clearing up the issue that our prior explanations have only mentioned biconnected component flipping, not connected component flipping. Consider processing the merge queue in order. For each pair w and w' , let w_{in} be one if w was entered from a counterclockwise direction and zero otherwise, and let w'_{out} be zero if w' was exited in a counterclockwise direction and one otherwise. Also, we augment the processing of the merge queue with a variable named *sign*, which is initially 1 and changes between 1 and -1 whenever a biconnected component flip occurs. For each 4-tuple $(w, w_{in}, w', w'_{out})$ pulled from the merge queue, the following steps are performed. First, if the sign is currently -1, then w_{in} is inverted (i.e. assigned the value 1 xor w_{in}) because the -1 sign indicates that the biconnected component containing w was flipped relative to when the 4-tuple was pushed onto the merge queue. Second, we flip the biconnected component rooted by w' if w_{in} and w'_{out} are equal. Since w_{in} and w'_{out} have opposite meanings, their equality signifies opposition in the directions of entry and exit. Third, if we flipped the biconnected component, then its value is reversed (assigned -1 if it is 1, or 1 if it is -1). Fourth, we merge the adjacency list of w' into the adjacency list of w between the two external face edges incident to w . Then, we reiterate these steps with the next 4-tuple from the merge queue until it is empty.

It is easy to see that this strategy performs flip operations only on biconnected components, yet it achieves the same results as the connected component flip. Although this strategy typically does less work, it can still do too much work over the entire embedding process. However, we needed the strategy so the flip operation could be restricted to biconnected components, and it is still a helpful introduction to the fully optimized version appearing in Section 5.

3.3 Example of Walkdown Processing

This section presents an example that demonstrates the key processing rules of the Walkdown routine. Figure 5(a) presents a partial embedding of a graph at the beginning of step v and with the following edges still to embed: (u, d) , (u, s) , (u, x) , (u, y) , (v, p) , (v, q) , (v, t) , (v, x) , and (v, y) . Note that the vertex i is inactive and the biconnected component rooted by w' is not pertinent. The square vertices are externally active. Now we will discuss the actions performed by the Walkdown to embed the back edges from v to its descendants.

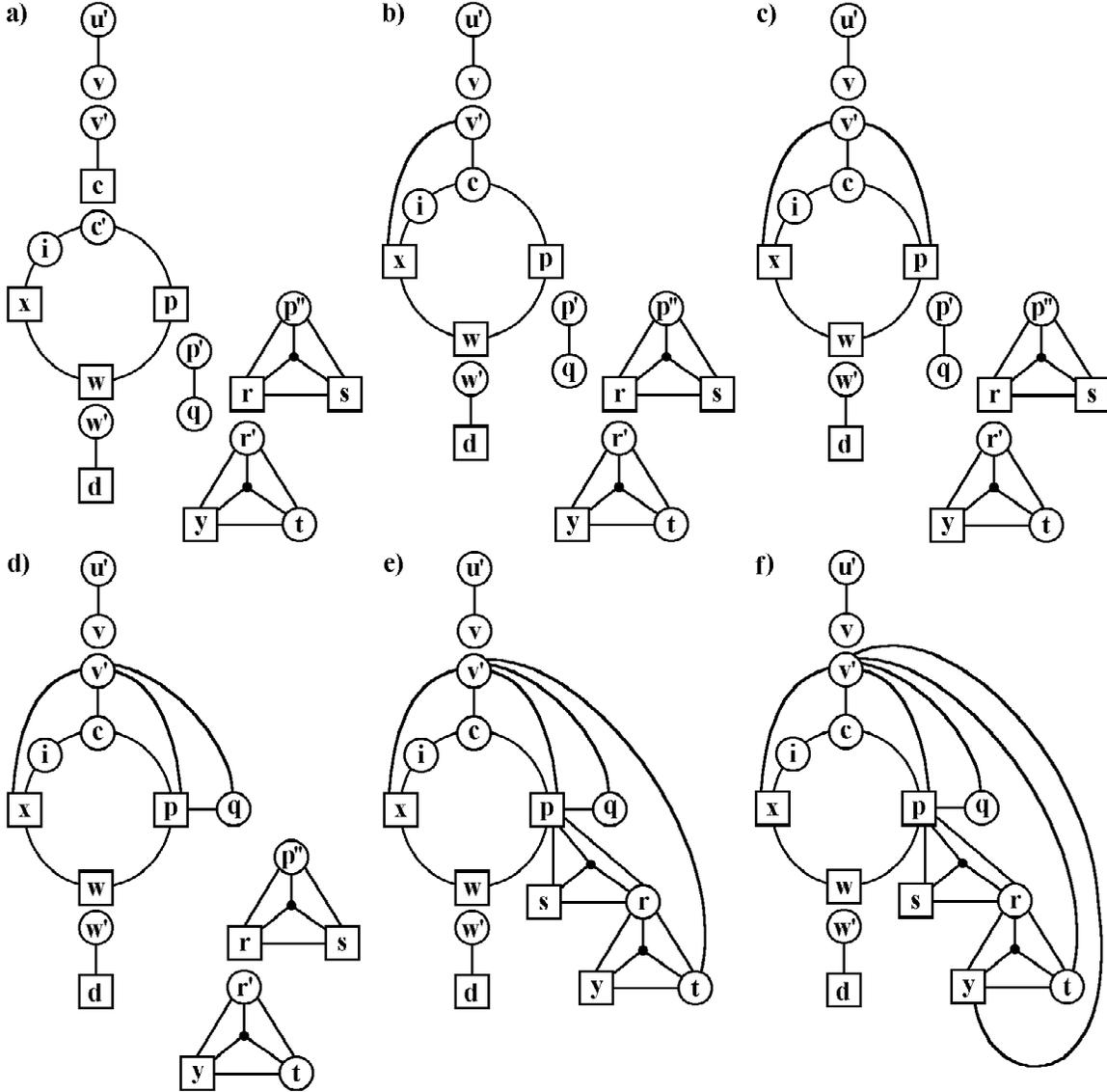


Figure 5: An Example of the Walkdown for Step v . Square vertices are externally active due to unembedded edges (u, d) , (u, s) , (u, x) , and (u, y) . Edges (v, p) , (v, q) , (v, t) , (v, x) , and (v, y) are to be added in step v . a) Embedding at the start of step v . b) Merge at c to add (v, x) , then stop counterclockwise traversal. c) Clockwise traversal visits p and embeds (v, p) . d) Merge p and p' and embed (v, q) . e) Flip biconnected component rooted by p'' , merge p and p'' , merge r and r' , then embed (v, t) . f) Embed (v, y) and stop clockwise traversal.

The first traversal in the counterclockwise direction begins at v' , travels to c , then de-

scends to c' . The first active vertices along the two external face paths are x and p , both of which are externally active and pertinent. The decision to proceed in the direction of x is therefore made arbitrarily. At x , there is a back edge to embed, so the Walkdown first merges c and c' with no flip operation since we happened to consistently travel counterclockwise when entering c and exiting c' . The result of the merge and the embedding of (v, x) appears in Figure 5(b).

Once the back edge to x has been embedded, the Walkdown determines that x has no pertinent child biconnected components and is therefore a stopping vertex that terminates the counterclockwise traversal. The second traversal commences in a clockwise direction from v' to c . Note that as soon as c and c' were merged in the first traversal, c was no longer externally active because it has no direct edge to an ancestor of v and it has no *separated* DFS child with a lowpoint less than v . Thus, it is now possible to proceed beyond c to p in the second traversal.

At p , the Walkdown first embeds the back edge (v, p) , the result of which is shown in Figure 5(c). Then, the Walkdown determines that p has pertinent child biconnected components. The Walkdown selects p' because it is the root of an internally active child biconnected component. The Walkdown descends to p' and finds that both paths lead to q , which is internally active. Next, p and p' are merged and the back edge (v, q) is embedded as shown in Figure 5(d). Since q becomes inactive as a result, the Walkdown proceeds from q to its successor on the external face, which is p .

In this second visitation of p , the Walkdown again tests whether a back edge to p must be embedded, but since the back edge has already been embedded, the result is negative. The Walkdown again tests for pertinent child biconnected components, but this time there are no internally active ones, so the Walkdown descends to p'' . The two external face paths from p'' lead to externally active vertices r and s , but r is pertinent and s is not, so the Walkdown proceeds in a counterclockwise direction from p'' to r (contrary to the clockwise direction by which the Walkdown entered p). At r , the Walkdown determines that there is no back edge to embed, but r does have a pertinent child biconnected component, so the Walkdown descends to r' . The two external face paths from r' lead to y and t . While y is pertinent, it is also externally active, whereas t is internally active. Thus, the clockwise path to t is selected, in opposition to the counterclockwise direction used to enter r .

At t , the Walkdown determines that a back edge must be embedded. Based on the first 4-tuple on the merge queue, the biconnected component rooted at p'' is flipped, the merge queue sign is changed to -1, and p'' merged with p . Now the second 4-tuple is pulled from the merge queue. Although the direction indicators associated with exiting r and entering r' were in opposition when the 4-tuple was created, the orientation of r has been inverted, which we detect by the merge queue sign of -1. Thus, the value of r_{in} is inverted, the biconnected component rooted at r' is not flipped, and r' is merged with r . Finally, the back edge (v, t) is embedded. The result of these operations is shown in Figure 5(e).

The clockwise traversal then continues from vertex t , which is now inactive, to vertex y . The back edge (v, y) is embedded as shown in Figure 5(f). Once the back edge to y is embedded, y is no longer pertinent since it has no pertinent child biconnected components. Thus, y is a stopping vertex that terminates the clockwise traversal of the Walkdown.

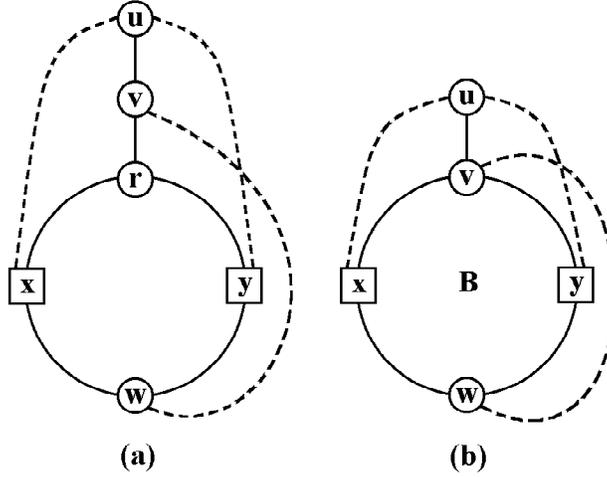


Figure 6: Walkdown Halting Conditions

4 Proof of Correctness

The Walkdown operations described in the example of the previous section embedded all of the back edges from v to its descendants in the given input graph, but consider what would happen if the input graph also contained the back edge (v, w) . Based on the processing rules, the Walkdown clearly cannot traverse to w , so the Walkdown would terminate without embedding (v, w) . For a given biconnected component B rooted by v^c , if the two Walkdown traversals embed all back edges between v and descendants of c , then it is easy to see that B remains planar and the algorithm continues. However, if some of the back edges to descendants of c are not embedded, then we show that the input graph is non-planar.

The Walkdown may halt if it encounters two stopping vertices while trying to determine the direction of traversal from the root of a pertinent child biconnected component. This condition is depicted in Figure 6(a). Otherwise, if the Walkdown halts on a biconnected component B without embedding all back edges to descendants of a virtual copy of v , then both Walkdown traversals were terminated by stopping vertices appearing along the external face of B . This condition is depicted by Figure 6(b).

In Figure 6(a), u represents all unprocessed ancestors of v , and (u, v) represents the DFS tree path from v to its ancestors. The edge (v, r) represents the path of descent from v to a pertinent child biconnected component rooted by a virtual copy of r . The Walkdown traversal is prevented from visiting a pertinent vertex w by stopping vertices x and y on both external face paths emanating from r . The cycle (r, x, w, y, r) represents the external face of the biconnected component. The dotted edges (u, x) , (u, y) and (v, w) represent connections from a descendant (x , y or w) to an ancestor (u or v) consisting of either a single unembedded back edge or a path containing a tree edge to a separated DFS child of the descendant, zero or more additional tree edges, and an unembedded back edge to the ancestor. Similarly, Figure 6(b) shows stopping vertices x and y that prevent traversal from reaching a pertinent vertex w in a biconnected component rooted by a virtual copy of v .

Both diagrams depict minors of the input graph. Since Figure 6(a) depicts a $K_{3,3}$, the input graph is non-planar. However, Figure 6(b) appears to be planar, so it is natural to ask why the edge (v, w) was not embedded inside B , which the Walkdown could do by embedding (v, w) along the external face, then embedding (v, x) such that (v, w) is surrounded inside the

bounding cycle of B . In short, there is either some aspect of the connection represented by edge (v, w) or some aspect of the vertices embedded within B that prevents the Walkdown from embedding the connection from w to v inside B . An examination of the possibilities related to these aspects yields four additional non-planarity minors, or five in total, which are depicted in Figure 7. Theorem 4.1 argues the correctness of our algorithm by showing that one of the non-planarity minors must exist if the Walkdown fails to embed a back edge, and the absence of the conditions that give rise to the non-planarity minors contradicts the assumption that the Walkdown failed to embed a back edge.

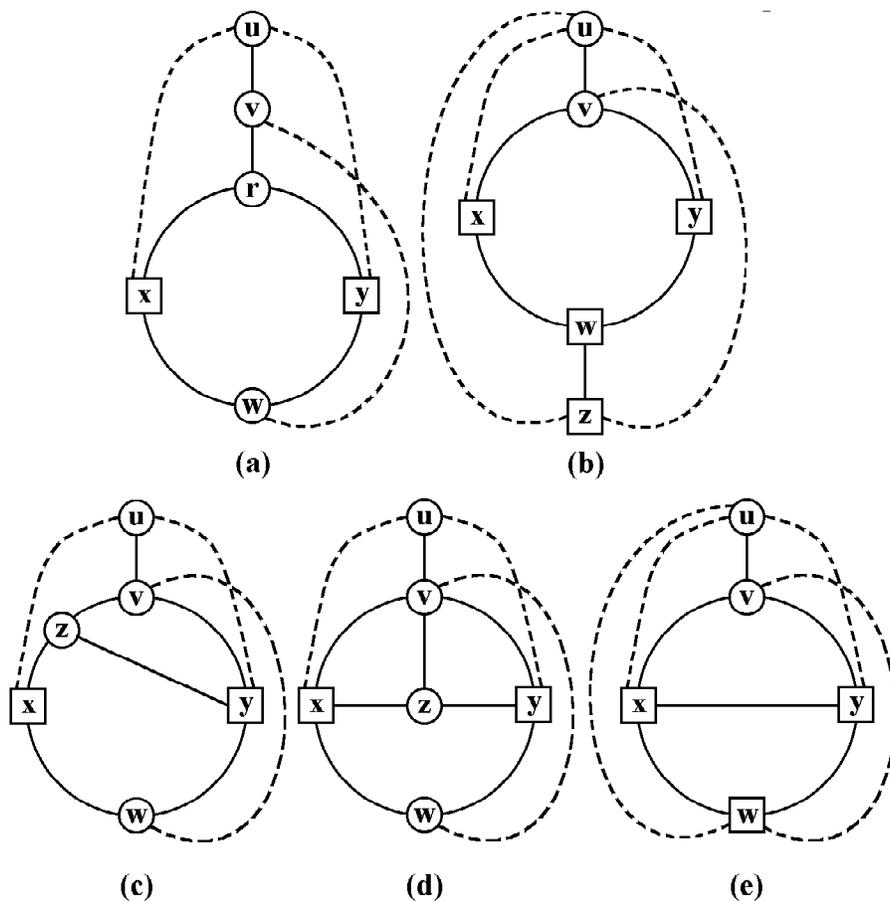


Figure 7: Non-planarity Minors of the Input Graph

Theorem 4.1 *Given a biconnected connected component B with root v^c , if the Walkdown fails to embed one or more back edges from v to descendants of c , then the input graph G is not planar.*

Proof. Figure 7(a) results if the merge queue is non-empty when either of the two Walkdown traversals is halted by a stopping vertex. The input graph is non-planar since Figure 7(a) is a $K_{3,3}$.

Figure 7(b) results if the pertinent vertex w has an externally active pertinent child biconnected component such that embedding the connection from w to v inside B would place an externally active vertex z inside B . If the Walkdown halts without embedding the

back edge that would clear the pertinence of w , then the input graph is non-planar since Figure 7(b) contains a $K_{3,3}$.

Otherwise we consider conditions related to having an obstructing path inside B that contains only internal vertices of B except for two points of attachment along the external face: one along the path v, \dots, x, \dots, w , and the other along the path v, \dots, y, \dots, w . The obstructing path, which is called an x - y path, contains neither v nor w . If such an x - y path exists, then the connection from w to v would cross it if the connection were embedded inside B . We use p_x and p_y to denote the points of attachment of the obstructing x - y path.

Figure 7(c) depicts the condition of having an x - y path in which p_x attached closer to v than x . Note that p_y can also be attached closer to v than y . In fact, Figure 7(c) also represents the symmetric condition in which p_y is attached closer to v than y (but p_x is attached at x or farther from v than x). In all of these cases, the input graph is non-planar since Figure 7(c) contains a $K_{3,3}$.

Figure 7(d) depicts the condition of having a second path of vertices attached to v that (other than v) contains vertices internal to B that lead to an attachment point z along the x - y path. If this second path exists, then input graph is non-planar since Figure 7(d) contains a $K_{3,3}$.

Figure 7(e) depicts the condition of having an externally active vertex (possibly distinct from w) along the lower external face path strictly between p_x and p_y . If this condition occurs, then input graph is non-planar since Figure 7(e) represents a K_5 minor.

Finally, suppose for the purpose of contradiction that the Walkdown has failed to embed a back edge and none of the non-planarity conditions described above exist. The merge queue must be empty due to the absence of the condition of Figure 7(a). By the contradictory assumption, a biconnected component rooted by a virtual copy of v has a pertinent vertex w along the lower external face path between stopping vertices x and y . We address two cases based on whether or not there is an obstructing x - y path.

If there is no obstructing x - y path, then at the beginning of step v all paths between x and y in the embedding contain w . Thus, w is a DFS ancestor of x or y (or both), and it becomes a merge point when its descendants (x or y or both) are incorporated into B . When the Walkdown first visits w , it embeds a direct back edge from w to v if one is required, so the pertinence of w must be the result of a pertinent child biconnected component. However, the Walkdown preferentially selects and processes internally active child biconnected components of w prior to attaching an externally active pertinent child biconnected component leading to x or y . Thus, the pertinence of w must be due to an externally active child biconnected component, which contradicts the pertinence of w since the condition of Figure 7(b) does not exist.

On the other hand, suppose there is an obstructing x - y path, but none of the remaining non-planarity minors apply. The *highest x - y path* is the x - y path that would be contained by a proper face cycle if the internal edges to v' were removed, along with any resulting separable components. At the beginning of step v , the highest x - y path and the lower external face path from p_x to p_y formed the external face of a biconnected component. Let r denote whichever of p_x or p_y had a virtual vertex that was the root of that biconnected component, and let s denote one of p_x or p_y such that $s \neq r$. Since the condition of Figure 7(c) does not exist, s is equal to or an ancestor of x or y and was therefore externally active when the Walkdown descended to r' . Moreover, when the Walkdown descended to r' , the first active vertex along the path that is now the highest x - y path is s because the condition of

Figure 7(d) does not exist. Descending from r' along the path that is now the lower external face path between p_x and p_y , the existence of a pertinent vertex w implies that there are no externally active vertices along the path due to the absence of the condition of Figure 7(e). Thus, we reach a contradiction to the pertinence of w since the Walkdown preferentially selects the path of traversal leading from the root of a child biconnected component to an internally active vertex. \square

Figure 8 exemplifies the conditions described by the final contradiction in the proof of Theorem 4.1 for the case $r = p_y$, $s = p_x$. In the example, $p_x \neq x$ and $p_y \neq y$ to promote clarity. At the beginning of step v , vertices x and y must have been externally active because they are stopping vertices for the Walkdown on v' . Note that p_x is depicted as an ancestor of x and y is an ancestor of p_y because the x - y path is attached low (if there were a high point of attachment, then Figure 7(c) would apply). We let w be the first active vertex along the clockwise external face path descending from p'_y , and we have assumed for the sake of contradiction that w is pertinent. Since Figure 7(d) does not apply, p_x is the first active vertex found along the counterclockwise external face path descending from p'_y . Finally, w is not externally active because Figure 7(e) does not apply. In this case, the Walkdown selects w as the next vertex to process, but the assumption that w is pertinent at the end of step v implies that p_x was selected.

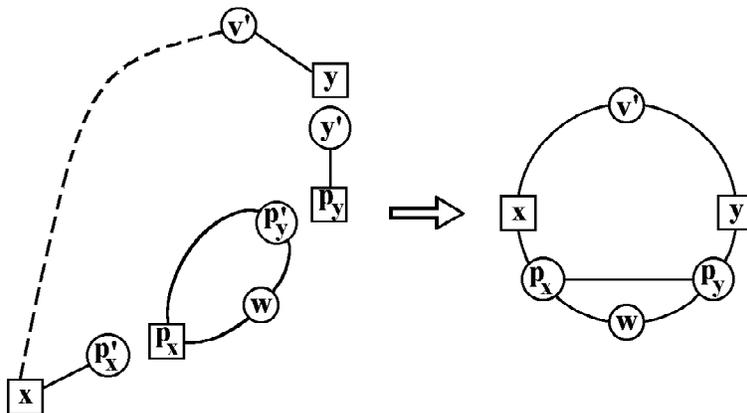


Figure 8: An Example for Theorem 4.1

5 Data Structures and Optimizations

5.1 DFS Parent and Lowpoint Calculations

Each vertex in \tilde{G} is represented by a vertex structure containing important information about the vertex such as its adjacency list. Non-virtual vertex structures contain a few members whose values are obtained during the preprocessing of the input graph G . The *DFSParent* member indicates the depth first search tree parent of a vertex (except that for a DFS tree root, the value *nil* since it has no parent). The *leastAncestor* indicates the ancestor with the least DFI adjacent to the vertex by a back edge in the input graph. The *lowpoint* of a vertex is the minimum of its leastAncestor and the lowpoint values of its DFS children. The lowpoint is a well-known and simple vertex parameter that can be computed by a post-order

traversal of the depth first search tree, or it can be computed during the initial depth first search [19].

5.2 Vertex Array and Virtual Vertices

In \tilde{G} , vertices (virtual and non-virtual) are represented by an array of $2n$ vertex structures. Using zero-based array indexing, vertices are stored in locations 0 to $n - 1$ and virtual vertices are stored in locations n to $2n - 1$. More specifically, a non-virtual vertex is stored at a location equal to its DFI, and a virtual vertex v^c is stored at location $n + c$. Thus, given a virtual vertex v^c , its non-virtual counterpart can be obtained in constant time by obtaining the DFSParent field in the vertex structure for c , where c is obtained by subtracting n from the location of v^c .

5.3 Externally Active Biconnected Components and Vertices

Given a virtual vertex w^c , it is possible to determine whether the biconnected component B containing w^c is externally active in constant time. First, we obtain c by subtracting n from w^c , then we test the lowpoint of c . If the lowpoint of c is less than v , then B is externally active because the DFS subtree rooted at c has at least one back edge connection to one or more ancestors of v .

To determine whether a given vertex w is pertinent, two additional members are added to each non-virtual vertex. The *separatedDFSCchildList* contains a circular doubly linked list of the DFS children of a vertex that appear in separate biconnected components. Prior to the embedding of any back edges, the *separatedDFSCchildList* of a vertex contains all of its DFS children, sorted by their lowpoint values. The sort can be performed in worst-case linear time by bucket sorting the vertices based on their lowpoint values, then adding each vertex in lowpoint order to the *separatedDFSCchildList* of its DFSParent. Each vertex receives a member named *repInParentList* to store a pointer to the representative of the vertex in the *separatedDFSCchildList* of its DFS parent. When the virtual vertex w^c is merged with its non-virtual counterpart w , the *repInParentList* member of the child c is used to remove c from the *separatedDFSCchildList* of its DFS parent w , which can be done in constant time since the *separatedDFSCchildList* is doubly linked.

A vertex w is externally active in step v of the main algorithm loop if its *leastAncestor* of w is less than v or if the first vertex in the *separatedDFSCchildList* of w has a lowpoint that is less than v . The first test determines whether w has an unembedded back edge directly to an ancestor of v , and the second test determines whether any child biconnected components of w are externally active.

5.4 Pertinence and the Walkup Routine

In Section 5.2, we discussed how to obtain the non-virtual counterpart of a given virtual vertex. However, it is too costly to maintain such a direct connection from a non-virtual vertex to all of its virtual vertices. Instead, at the beginning of each step v of the main algorithm loop (see Figure 4), the algorithm computes a list of virtual vertices that must be merged with their respective non-virtual counterparts during the embedding of the back edges from v to its descendants. Each non-virtual vertex has a circular doubly linked list called *pertinentBicompList* that is initially empty. When a pertinent child biconnected component

containing the virtual vertex w^c is found, then w^c is added to the `pertinentBicompList` of w . When w^c is merged with w , w^c is removed from the `pertinentBicompList` of w (in constant time). At the end of step v , all `pertinentBicompList` members are returned to the empty state, except of course if the Walkdown halts with a non-planarity condition.

When a virtual vertex w^c is added to the `pertinentBicompList` of w , it is either prepended or appended. Let B denote the pertinent child biconnected component containing w^c . If B is externally active, then w^c is appended; otherwise w^c is prepended. Since the internally active child biconnected components appear first in the `pertinentBicompList`, the Walkdown can preferentially select an internally active child biconnected component, if one is available, in constant time.

To complete the efficient implementation of the definition of pertinence, each non-virtual vertex has an additional member called *adjacentTo* and each vertex structure (including those for virtual vertices) has a member called *visited*. All *visited* and *adjacentTo* members are initially set equal to n and are used like flags. In a step v , the flag is set if it is equal to v and clear otherwise. The *adjacentTo* flag of a descendant w of v is set equal to v at the beginning of step v if there exists a back edge (v, w) in the input graph. The *adjacentTo* flag of w is cleared as soon as the back edge is embedded (so the Walkdown does not embed the edge again if it revisits w). All *visited* flags are cleared when the main algorithm loop decrements v . The *visited* flag is only set by the Walkup routine described below.

The *Walkup* routine is invoked at the beginning of step v once for each back edge (v, w) in the input graph, where w is a DFS descendant of v . It first sets the *adjacentTo* flag of w . Then, the Walkup performs a loop that begins at w and simultaneously traverses both paths originating from w around the external face of the biconnected component B_w containing w until the root vertex r' of B_w is encountered. If r' is a virtual copy of v , then the Walkup terminates successfully. Otherwise, r' is stored in the *pertinentBicompList* of the non-virtual vertex structure r . The virtual copy is appended if B_w is externally active and prepended otherwise. Then, the Walkup loop reiterates starting at r .

Within each biconnected component visited by Walkup, one of the two external face paths between r' and w becomes part of a new proper face once the back edge (v, w) is embedded. The Walkup performs external face traversal in parallel to ensure that each biconnected component root is found with a cost not exceeding twice the size of the shortest path around the external face. This helps to ensure that the total cost of all Walkup operations is a constant factor of the sum of degrees of proper faces in the embedding.

The Walkup loop also sets the *visited* member of each virtual and non-virtual vertex it encounters. Any future Walkup invocation in step v terminates immediately if it encounters a visited vertex structure. The purposes of this optimization is to ensure that the cumulative work done by all Walkup calls for the back edges of v does not exceed a constant factor of the number of bounding edges in new proper faces formed during step v .

5.5 Short-circuit Edges

Unlike the Walkup, the Walkdown cannot simply choose the shortest path when it descends to a biconnected component root r' and selects a path to the next vertex to visit. The Walkdown must find the first active vertex along both external face paths emanating from r' , regardless of their length. It is possible to create input graphs on which the Walkdown traverses an $O(n)$ length path $O(n)$ times. To rectify this problem, the Walkdown can be

modified to eliminate the paths of inactive vertices such that the immediate neighbors of a virtual vertex along the external face are always active.

When the Walkdown is visiting a vertex w , it may be inactive or may become inactive after adding a back edge to w . The Walkdown simply obtains the successor s along the external face and reiterates, visiting s . Before the Walkdown reiterates, we augment the processing of an inactive vertex w by adding a special ‘short-circuit’ edge between s and the root v' of the biconnected component B , removing w from the external face. However, to ensure that we do not exceed the total edge limit of $3n - 5$, the short-circuit edge is not embedded if the adjacentTo member of s equals v , if B is not externally active, or if B does not contains s . Finally, since each short-circuit edge is specially marked, they are easy to remove after the main loop in Figure 4 is finished.

5.6 Edge Representation

Each edge (u, v) is represented by a pair of edge records that are inserted into the adjacency lists of u and v . The edge record in the adjacency list of u indicates v as a *neighbor* and vice versa. The neighbor field of an edge record indicates either a virtual or non-virtual vertex. An edge record also carries a *type* member to indicate whether it is part of a tree edge, back edge or short-circuit edge. Each edge record has a *sign* member that is initially 1 but changed to -1 in a tree edge (w^c, c) if the biconnected component rooted by w^c must be flipped before w^c is merged with w .

The edge records for all edges are stored in an array of size $k(3n - 5)$ for any constant $k \geq 2$. The two edge records representing an edge are stored at consecutive locations such that traversal of an edge in either direction is a constant time operation. Given the position p of an edge record, the associated edge record is at position $p + 1$ if p is even or at $p - 1$ if p is odd. We refer to this calculated connection between edge records as the *twin link*.

5.7 Adjacency Lists and Maintaining the External Face

The adjacency list of each virtual and non-virtual vertex is a doubly linked circular list that include the vertex structure plus the list of edge records indicating each neighbor. Thus, each vertex structure and edge record contains two pointers, denoted *link[0]* and *link[1]*. Moreover, since an edge record link can indicate either another edge record or a vertex structure, it is necessary to be able to identify the type of object at which a link points. One way to do this is to create a common structure capable of representing either a vertex structure or edge record. Then, a single array can be used to store first the $2n$ vertex structures then the edge records, the links can be represented as indices into the array, and edge records can be distinguished by having a location of $2n$ or greater.

The purpose of linking a vertex structure into its adjacency list is two-fold. Firstly, if the vertex is on the external face, then the vertex structure’s *link[0]* and *link[1]* pointers indicate edge records of edges that join the vertex to the bounding cycle of the external face. Secondly, if we have an edge record of an edge on the bounding cycle of the external face, then it is possible to obtain the vertex whose adjacency list contains the edge record because either *link[0]* or *link[1]* points to a vertex structure (or both if the edge is the only one in a singleton biconnected component).

5.8 Merging and Flipping a Child Biconnected Component

In Section 1, a combinatorial planar embedding is defined to provide a clockwise ordered list of the neighbors of each vertex. Since it is easy to create graphs that would require $O(n)$ vertices to be inverted $O(n)$ times, we cannot afford to directly maintain a consistent vertex orientation throughout the embedding process. Our solution is to relax the requirement that a consistent vertex orientation be maintained during embedding. Our data structures maintain a cyclic order of the embedded edges of each vertex, but individual vertices can have a clockwise or counterclockwise orientation. Technically, we do not violate the definition since a consistent orientation for all vertices is well-defined and easy to recover at any time using a depth first search within each biconnected component.

Since we need only maintain the cyclic edge order of each vertex, the biconnected component flip operation can be reduced to a simple augmentation of our process for merging a virtual vertex with its non-virtual counterpart. When the Walkdown descends from a vertex w to a virtual vertex w' , four integers are pushed into the merge queue, which are denoted w , w_{in} , w' and w'_{out} . The meaning of w and w' have already been explained. The link in w indicating the edge record the Walkdown used to enter w is denoted w_{in} , and w'_{out} denotes the link in w' indicating the edge record the Walkdown used to exit w' toward the next vertex. When the Walkdown finds a back edge to embed, it embeds the edge, but it also processes the queue to merge all biconnected components it has encountered since the last edge embedding. The merge of w and w' must occur such that the edge records indicated by w_{in} and w'_{out} become consecutive in the adjacency list of w and such that they lie on the proper face created by the back edge being embedded.

Merging w^c with w consists of the following operations. First, w^c is removed from the head of the pertinentBicompList of w , and c is removed from the separatedDFSChildList of w . Then, all settings in the edge records of edges incident to w^c are changed to indicate incidence with w . Then, a circular list union of the adjacency lists of w and w' occurs. The link[w_{in}] edge record of w and the link[w'_{out}] edge record of w' are joined, and the link[1 xor w'_{out}] edge record of w' becomes the new link[w_{in}] edge record of w .

The biconnected component flip operation occurs implicitly as a part of correctly performing the circular list union. Note that if w_{in} and w'_{out} are equal, then the links in each edge record of the adjacency list of w' must be swapped before the join operations described above can result in a consistent adjacency list. The swapping of the links in each adjacency list node inverts the orientation of w' to be consistent with w , but the orientations of all descendants of w' are not changed (because it would take too long). Instead, the root edge incident to w' is marked with a sign of -1 so that a post-processing operation described below can recover the proper orientation of the descendants of w' . Note that the flipping operation implicitly inverts the orientation of all descendants of w' , not just the vertices in the biconnected component with root w' .

The details of these processes are illustrated in Figures 9, 10, and 11. In Figure 9, we have an overview of the embedding of back edge (1, 7). Figure 9(a) shows the state of the data structures during step 1 after embedding back edges (1, 3) and (1, 4). Because vertex 4 is externally active, the first Walkdown traversal returns and the second Walkdown traversal begins at $1'$ such that back edge (1, 7) will be embedded around the right hand side. However, since vertex 8 is also externally active, the biconnected component rooted at $2'$ must be flipped so that vertex 8 remains on the external face when edge (1, 7) is embedded. The result is shown in Figure 9(b).

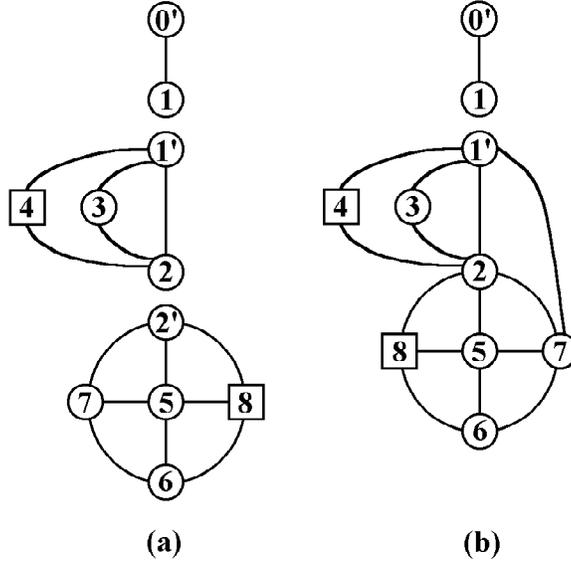


Figure 9: Overview of Data Structures for Flip Operation

An elaboration of Figure 9(a) appears in Figure 10. The rounded rectangles are edge records, and the double lines connecting them are twin links. The circles represent vertex structures, except that vertices 4 and 8 are represented by squares to indicate their external activity. The single lines with black and white dots for endpoints represent the links that bind the adjacency list and the vertex structure into a circular doubly linked list.

At this point of the embedding, all vertices still have the same orientation. The edge records in the adjacency list of any vertex can be traversed in counterclockwise order by traversing the black dot links to exit the vertex structure and each edge record in the adjacency list.

As stated previously, the first Walkdown traversal embeds edges $(1,3)$ and $(1,4)$, then stops at vertex 4. The second Walkdown traversal restarts along the right side of edge $(1',2)$. Then, it descends to the pertinent child biconnected component rooted at $2'$. Since vertex 7 is internally active, the biconnected component must be flipped before merging so that vertex 8 remains on the external face.

The results of the merge operation can be seen in Figure 11. The merge operation begins by changing all edge records that contain $2'$ so that they contain 2. Then, we flip the biconnected component by inverting the circular list union. Edge record $(2,7)$ is joined with $(2,1)$, and edge record $(2,8)$ is joined with vertex structure 2. Note that the links of the edge records formerly in $2'$ must be inverted in this case so that the adjacency list of vertex 2 is consistent, i.e. a traversal of the adjacency list of vertex 2 can be performed by consistently using the same colored dot link to exit the vertex structure and each edge record.

The inverted circular list union has, however, implicitly inverted the orientations of all of the descendants of vertex $2'$. No other work was actually performed on the links in these descendants, and the result is that the black dots links now result in a clockwise ordering of their adjacency lists. Our algorithm accounts for this by marking the tree edge $(2,5)$ with a sign of -1.

The final change made in Figure 11 is the addition of the back edge $(1,7)$. Since we exited vertex $1'$ using edge record $(1',2)$, the new edge record $(1',7)$ is added between vertex

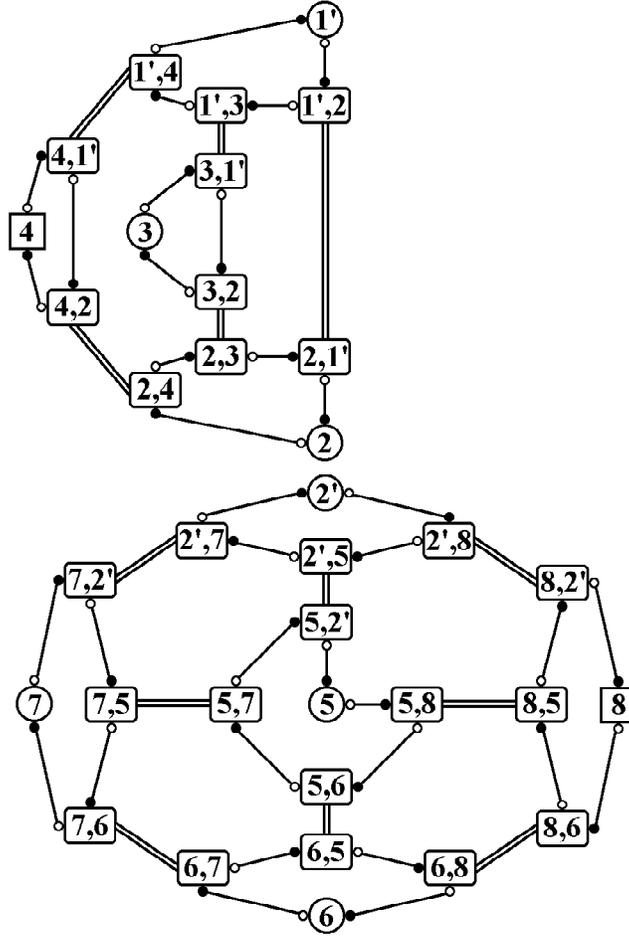


Figure 10: Elaboration of Data Structures Before Flip Operation

structure $1'$ and edge record $(1', 2)$. Since the Walkdown entered vertex 7 using the edge record $(7, 2')$, the new edge record $(7, 1')$ is added between the vertex structure for 7 and the edge record for $(7, 2)$. Thus, edge $(1, 7)$ is on the external face and has formed a new proper face in the embedding that includes vertices 1, 2, and 7.

Based on the detail in Figure 11, it is evident that our strategy of biconnected component flipping introduces a small wrinkle in how we traverse the external face of biconnected components. A counterclockwise walk of the external face of the biconnected component now rooted at $1'$ begins with vertex $1'$ then vertices 4, 2, 8, 6, 7 and back to $1'$. In detail, we exit vertex $1'$ using a black dot link, then we exit vertex 4 using its black dot link, then we exit vertex 2 using a black dot link. This is because vertices $1'$, 4 and 2 have the same orientation. However, if we exit vertex 8 using the black dot link, this takes us back to vertex 2. This is because vertex 8 has an opposing orientation as described above.

To solve this problem, we switch focus from the link used to exit a vertex w and instead maintain the link used to enter its successor s (which is determined specially for singleton biconnected components, as described in Section 5.9). Then, our algorithms simply exit a vertex s from the opposing link. In the case of vertex 8 in Figure 11, we enter from the edge record $(8, 2)$, which corresponds to the black dot link for vertex 8, so we exit from the white dot link of vertex 8. Similarly, we reach vertices 6 and 7 through black dot links, so we exit each through their respective white dot links, which returns properly to vertex $1'$.

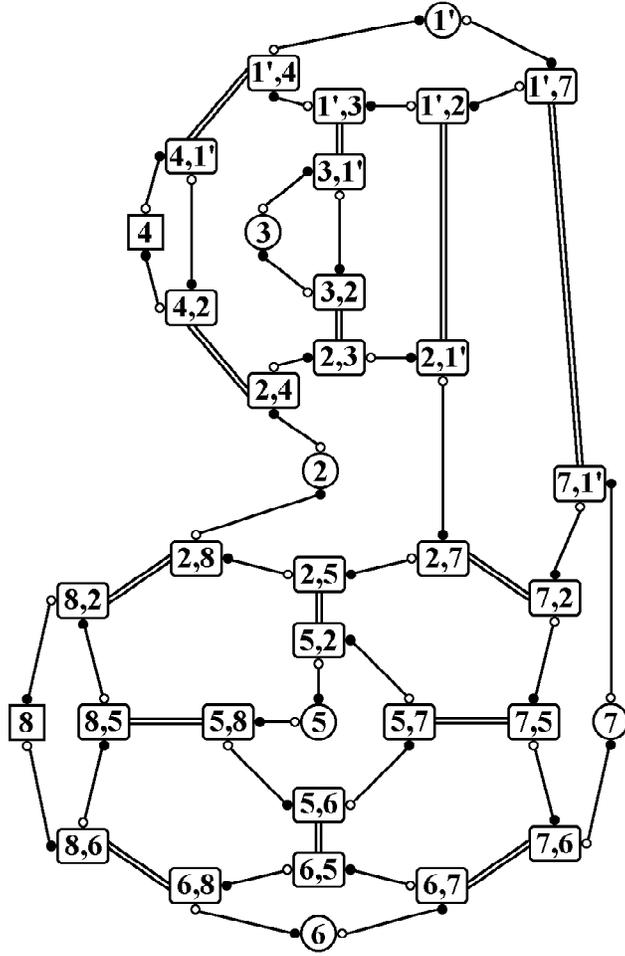


Figure 11: Data Structures After Flip and Back Edge Embedding

5.9 Traversing the External Face

Due to our biconnected component ‘flip’ strategy, an orientation for a vertex w that is consistent with the orientation of the root r' of the biconnected component containing w is obtained by inverting w if the product of the signs of the edges on the tree path from w to r is -1 . This optimization does not technically violate the definition of combinatorial planar embedding because consistent vertex orientations are well-defined and easily recoverable by a depth first search of each biconnected component. However, the method of traversing the external face of a biconnected component becomes slightly more involved than a simple traversal of a cycle of vertices and edges.

During a traversal, suppose we have arrived at a vertex w from an edge containing the edge record indicated by the $\text{link}[w_{in}]$ member of w (where w_{in} is 0 or 1), and suppose we would now like to travel to the successor s of w on the external face. Let e denote the edge record indicated by the $\text{link}[w_{out}]$ member of w , where $w_{out} = 1xorw_{in}$, and let e_{twin} denote the edge record associated with e by the twinLink . Either $\text{link}[0]$ or $\text{link}[1]$ of e_{twin} indicates a vertex structure, which represents the successor vertex s . Finally, let s_{in} be equal to 0 or 1 such that $\text{link}[s_{in}]$ in s indicates e_{twin} (or let s_{in} equal w_{in} if s is degree one). Note that we track the 0 or 1 link of an edge record and not the edge record itself because it simplifies the task of keeping track of the direction of travel while descending through biconnected

components containing a single tree edge.

5.10 Recovering an Embedding

After the main algorithm loop (Figure 4), a simple post-processing operation can be used to impose a consistent orientation on all vertices in each biconnected component of the embedding. The orientation of a vertex w relative to the root of the biconnected component is determined by the product of the signs of the tree edges in the DFS tree path from w up to the root of the biconnected component. If the product equals -1, then w must be inverted by swapping the links in its vertex structure and in each edge record of its adjacency list. The required product for each vertex can be computed in an orderly fashion using a pre-order traversal of the portion of the depth first search tree in the biconnected component.

Once each biconnected component is oriented, any remaining virtual vertices are merged with their non-virtual counterparts (with no flipping operations). Virtual vertices in the final embedding are representative of DFS tree roots and cut vertices.

6 Linear Time Implementation

6.1 Walkup Pseudo-code

At the beginning of each step v , the Walkup routine is invoked once per back edge of G that is incident to v and a descendant w . As a result of all Walkup invocations, all pertinent vertices and biconnected components are identified. Pseudo-code for the Walkup appears in Figure 12, which is an elaboration of the Walkup description given in Section 5.4.

The `adjacentTo` flag in the descendant vertex w is set in Line 1 of the pseudo-code in Figure 12. A *traversal context* consists of a vertex and a direction indicator, which is a 0 or 1 identifying whether the vertex was entered by the `link[0]` or `link[1]` edge record. Lines 2 and 3 initialize two traversal contexts for the simultaneous walk around the external faces of biconnected components. Since the simultaneous traversal is being initialized to start with the biconnected component containing w , both traversal contexts start at vertex w and are assigned opposing direction indicators.

The loop beginning in Line 4 of the pseudo-code in Figure 12 terminates if traversal reaches v or, according to Line 5, if a prior Walkup in step v has visited either of the vertices x or y indicated by the two traversal contexts. If a prior Walkup in step v has visited x or y , then the remaining virtual vertices on the ancestor path from x or y up to v have already been recorded in the `pertinentBicompList` members of their non-virtual counterparts. If neither x nor y have been visited by the Walkup in step v , then they are marked as visited in Line 6. Note that this includes marking virtual vertices as visited.

Line 7 to 10 of the pseudo-code in Figure 12 determine whether either traversal context has found the virtual vertex at the root of the biconnected component being traversed. If so, then Lines 11 to 18 add the virtual vertex, denoted z' , to the `pertinentBicompList` of its non-virtual counterpart z and then transfer both traversal contexts to z . Given z' , lines 11 and 12 determine the location of z according to the method described in Section 5.2. Line 13 ensures that the virtual vertex z' is not added to the `pertinentBicompList` of z if z is equal to v because v does not become a merge point in step v . Line 14 tests whether the biconnected component rooted by z' is externally active according to the method described

Figure 12: The Walkup Routine

Procedure: Walkup

this: Embedding Structure \tilde{G}

in: A vertex w (a descendant of the current vertex v being processed)

- (1) Set the adjacentTo member of w equal to v
- (2) $(x, x_{in}) \leftarrow (w, 1)$
- (3) $(y, y_{in}) \leftarrow (w, 0)$
- (4) while $x \neq v$
- (5) if the visited member of x or y is equal to v , break the loop
- (6) Set the visited members of x and y equal to v
- (7) if x is a virtual vertex, $z' \leftarrow x$
- (8) else if y is a virtual vertex, $z' \leftarrow y$
- (9) else $z' \leftarrow nil$
- (10) if $z' \neq nil$
- (11) $c \leftarrow z' - n$
- (12) Set z equal to the DFSParent of c
- (13) if $z \neq v$
- (14) if the lowpoint of c is less than v
- (15) Append z' to the pertinentBicompList of z
- (16) else Prepend z' to the pertinentBicompList of z
- (17) $(x, x_{in}) \leftarrow (z, 1)$
- (18) $(y, y_{in}) \leftarrow (z, 0)$
- (19) else $(x, x_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(x, x_{in})$
- (20) $(y, y_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(y, y_{in})$

in Section 5.3. Lines 15 and 16 ensure that internally active biconnected components appear earlier in a pertinentBicompList than those that are externally active. Finally, Lines 17 and 18 restart the simultaneous external face traversal on the biconnected component containing z (or Line 4 terminates the process if z equals v).

On the other hand, if Lines 7 to 10 of the pseudo-code in Figure 12 determine that neither traversal context found the biconnected component root, then Lines 19 and 20 simply advance both traversal contexts to the next vertices along the external face by invocations of a simple low-level routine called GetSuccessorOnExternalFace, which implements the traversal logic described in Section 5.9.

Lemma 6.1 *The Walkup routine achieves constant amortized time per vertex of the input graph G .*

Proof. Every line of the Walkup pseudo-code in Figure 12 expresses a constant time operation. The loop in Lines 4 to 20 operates over the shorter external face path from a starting vertex to the root of each biconnected component visited. Since each Walkup terminates as soon as it encounters a path traversed by a prior Walkup in step v , the constant work of the loop body in Lines 5 to 20 is applied to paths that are no longer than the degrees of the proper faces formed as the back edges are embedded that initiated the Walkup invocations. If the Walkdown fails to embed a back edge in step v , then the planarity algorithm halts in step v , so the cost of the Walkup invocations in step v can be associated as a one-time additional $O(n)$ cost. \square

6.2 Walkdown Pseudo-code

Once all Walkup invocations have been performed in step v , the Walkdown routine is invoked once per tree edge of G that is incident to v and a child c . The Walkdown embeds each back edge from v to a descendant of c . Pseudo-code for the Walkdown appears in Figure 13, which is an elaboration of the Walkdown description given in Section 3.2.

In Line 1 of the pseudo-code in Figure 13, the merge queue is created (or cleared if the implementation creates it outside of the Walkdown routine). Line 2 iterates over the remaining pseudo-code to perform the counterclockwise and clockwise edge embedding traversals. Line 3 obtains the successor of v' based on the direction selected by the Line 2 loop variable v'_{out} . Line 4 begins a loop that advances the traversal context (w, w_{in}) until it returns to v' , though the loop may be stopped before this occurs, as described below.

When visiting a vertex w , the first task is to determine whether a back edge should be embedded between w and v' , which is tested by Line 5 of the pseudo-code in Figure 13. If so, then Lines 6 to 9 merge biconnected components and embed the back edge. Specifically, Line 6 iterates while the merge queue is not empty, and Line 7 invokes a routine that pulls each 4-tuple $(r, r_{in}, r', r'_{out})$ from the merge queue and performs the merge of r and r' described in Section 5.8, which includes the flip operation if needed as well as removing the necessary elements from the pertinentBicompList and separatedDFSCchildList of r . Once this is done, Line 8 embeds the back edge (v', w) such that its edge records are indicated by $\text{link}[v'_{out}]$ in v' and $\text{link}[w_{in}]$ in w . Finally, Line 9 clears the adjacentTo flag of w so that another back edge to w is not embedded if the Walkdown revisits w .

Once the back edge embedding task is done, the second task of the Walkdown is to determine whether a vertex w has any pertinent child biconnected components to be processed, which is tested in Line 10 of the pseudo-code in Figure 13. If so, the Lines 11 to 21 push a new 4-tuple onto the merge queue and descend the traversal to a child biconnected component of w . Line 11 pushes vertex w and the direction of entry w_{in} onto Q . Line 12 selects the first element of the pertinentBicompList of w , which is guaranteed to be the root of an internally active child biconnected component if any exist in the list (i.e. the roots of internally active child biconnected components are at the beginning of the pertinentBicompList because they are prepended by the Walkup). Lines 13 and 14 create two traversal contexts to find the first active vertex along both external face paths emanating from w' . When short-circuit edges are used, the desired active vertices are guaranteed to be neighbors of w' . Lines 15 to 18 implement the decision logic about which path to take from w' , and Lines 19 to 21 pushes w' and an indicator w'_{out} of the chosen direction to the next vertex to be visited.

Figure 13: The Walkdown Routine

Procedure: Walkdown

this: Embedding Structure \tilde{G}

in: A virtual vertex v' associated with DFS child c

- (1) Create an empty merge queue Q
- (2) for v'_{out} in $\{0, 1\}$
- (3) $(w, w_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(v', 1 \text{ xor } v'_{out})$
- (4) while $w \neq v'$
- (5) if the adjacentTo member of w is equal to v ,
- (6) while Q is not empty,
- (7) MergeBiconnectedComponent(Q)
- (8) EmbedBackEdge(v', v'_{out}, w, w_{in})
- (9) Set the adjacentTo member of w equal to n
- (10) if the pertinentBicompList of w is non-empty,
- (11) Push (w, w_{in}) into Q
- (12) $w' \leftarrow$ value of first element of pertinentBicompList of w
- (13) $(x, x_{in}) \leftarrow \text{GetActiveSuccessorOnExternalFace}(w', 1)$
- (14) $(y, y_{in}) \leftarrow \text{GetActiveSuccessorOnExternalFace}(w', 0)$
- (15) if x is internally active, $(w, w_{in}) \leftarrow (x, x_{in})$
- (16) else if y is internally active, $(w, w_{in}) \leftarrow (y, y_{in})$
- (17) else if x is pertinent, $(w, w_{in}) \leftarrow (x, x_{in})$
- (18) else $(w, w_{in}) \leftarrow (y, y_{in})$
- (19) if w equals x , $w'_{out} \leftarrow 0$
- (20) else $w'_{out} \leftarrow 1$
- (21) Push (w', w'_{out}) into Q
- (22) else if w is inactive,
- (23) $(w, w_{in}) \leftarrow \text{GetSuccessorOnExternalFace}(w, w_{in})$
- (24) if the lowpoint of $c = v' - n$ is less than v
and the adjacentTo member of w is not equal to v ,
- (25) EmbedShortCircuitEdge(v', v'_{out}, w, w_{in})
- (26) else break the ‘while’ loop
- (27) if Q is non-empty, break the ‘for’ loop

If, on the other hand, Line 10 determined that the vertex w had no pertinent child biconnected components, then w is not pertinent since a back edge to w was embedded by

Lines 5 to 9 if needed. Thus, w is either inactive or externally active. A vertex is externally active if its leastAncestor member is less than v or if the lowpoint of the first child in its separatedDFSCildList is less than v (the list is sorted by lowpoint, so the first child has the least lowpoint). If Line 22 determines that w is inactive, then Line 23 obtains the successor along the external face, and Lines 24 and 25 embed a short-circuit edge if one is required. Note that an inactive vertex is only encountered if it is in the same biconnected component as v' , so a test to ensure that Q is empty is not needed. The first test of Line 24 ensures that the biconnected component containing the inactive vertex is externally active so that if the addition of a short-circuit edge results in having only two edges along the external face, then the face is guaranteed to be bisected in a later step. The second test in Line 24 ensures that a short-circuit edge will not be embedded if the next loop iteration will embed a back edge. Thus, the use of short-circuit edges does not result in any faces of degree 2 in the final planar embedding (so the planar graph edge limit is not violated by the use of short-circuit edges). If these criteria are met, then Line 25 embeds the short-circuit edge in the same manner as a back edge except that it is marked as a short-circuit edge to facilitate removal after the algorithm finishes embedding edges.

Finally, if the vertex w being visited is neither pertinent nor inactive, then it must be an externally active stopping vertex, so Line 26 stops the traversal for the direction indicated by v'_{out} . Line 27 breaks the outer loop, potentially stopping it from making the second traversal for $v'_{out} = 1$ if a stopping vertex was encountered on Line 26 when Q was non-empty. It is not strictly necessary to perform this action, but it is useful if Q must be preserved and returned by the Walkdown, as is the case for Kuratowski subgraph isolation (see Section 7).

Lemma 6.2 *The Walkdown routine achieves constant amortized time per vertex of the input graph G .*

Proof. Each line of the Walkdown pseudo-code in Figure 13 expresses a constant time operation except Line 7, 12 and 13. In Line 7, merging a virtual vertex performs constant work on the virtual vertex and its non-virtual counterpart plus constant work per edge incident to the virtual vertex. Since a merge never performs work on a virtual vertex or an edge more than once, the total cost is $O(n)$, or constant amortized time per vertex. As for Lines 12 and 13, each desired vertex is found after traversing a single edge due to short-circuit edges embedded in prior steps (greater than v). Thus, the while loop in Lines 4 to 26 achieves constant amortized time in each iteration. Each time a back edge or short-circuit edge is embedded, the number of iterations performed since the last edge was embedded or since the loop started is limited to the length of the cycle bounding the proper face that is formed when the new edge is added. Thus, the total cost of all while loop iterations is $O(n)$, or constant amortized time per vertex, except the iterations that occur after the last edge is embedded by the while loop. Since the while loop finds a stopping vertex in at most one iteration after the last edge is embedded, the cost of finding stopping vertices can be associated with the tree edge on which the Walkdown was invoked. If the while loop stops due to the condition of Line 4, then the biconnected component being processed has become inactive, so the cost of traversing its external face can be charged as a one-time extra cost per vertex and edge on the external face (an inactive biconnected component is never visited again by the Walkdown). Finally, additional constant time costs of Lines 1 to 3 and 27 can be associated with the tree edge on which the Walkdown was invoked. \square

6.3 Planarity by Edge Addition

This section presents the top-level routine of our new planarity algorithm. The pseudo-code in Figure 14 is an elaboration of the high-level algorithm in Figure 4 that can achieve $O(n)$ performance using the data structures and optimizations in Section 5 as well as the optimized versions of Walkup and Walkdown appearing in Sections 6.1 and 6.2.

Figure 14: The Edge Addition Planarity Algorithm

Procedure: Planarity

in: Simple undirected graph G with $n \geq 2$ vertices and $m \leq 3n - 5$ edges

out: An indication of whether or not G is planar and an embedding structure

\tilde{G} containing either a planar embedding of G or a Kuratowski subgraph of G .

- (1) Perform depth first search and lowpoint calculations for G
- (2) Create and initialize \tilde{G} based on G , including creation of separatedDFSCChildList for each vertex, sorted by child lowpoint
- (3) for each vertex $v \in G$
- (4) for each DFS child c of v in G ,
- (5) Embed tree edge (v^c, c) in \tilde{G}
- (6) for each vertex v from $n - 1$ to 0 in descending order
- (7) for each back edge of G incident to v and a descendant w
- (8) $\tilde{G}.$ Walkup(w)
- (9) for each DFS child c of v in G
- (10) $v' \leftarrow c + n$
- (11) $\tilde{G}.$ Walkdown(v')
- (12) for each back edge of G incident to v and a descendant w
- (13) if the adjacentTo member of w in \tilde{G} equals v
- (14) $\tilde{G}.$ IsolateKuratowskiSubgraph(G, v)
- (15) return (NONPLANAR, \tilde{G})
- (16) Delete short-circuit edges
- (17) for each biconnected component B rooted by a virtual vertex r'
- (18) Impart consistent orientation to vertices in B
- (19) Merge each remaining virtual vertex with its non-virtual counterpart
- (20) return (PLANAR, \tilde{G})

In the pseudo-code of Figure 14, Line 1 performs the depth first search that assigns a depth first index (DFI) and depth first search parent to each vertex and that partitions the edges into tree edges and back edges. Line 2 creates a data structure \tilde{G} with $2n$ vertex structures and $3n - 5$ pairs of edge records. The members of vertex structures and edge records are initialized in the manner described in Section 5. Lines 3 to 5 embed in \tilde{G} a

singleton biconnected component for each tree edge of G . Lines 6 to 16 embed the back edges (or discover that G is not planar). Line 6 begins a loop that iterates through each vertex in reverse DFI order. Lines 7 and 8 invoke the Walkup routine of Section 6.1 for each back edge from v to a descendant w . Lines 9 to 11 invoke the Walkdown routine of Section 6.2 for each tree edge from v to a child c . After completing all Walkdown invocations, Lines 12 and 13 test for non-planarity by determining if the Walkdown failed to embed any back edges. If so, then Lines 14 and 15 isolate a Kuratowski subgraph with a routine described in Section 7 and return the result along with an indication that G is not planar. On the other hand, if the loop in Lines 6 to 15 of the pseudo-code in Figure 14 embeds all back edges of G in \tilde{G} , then a few post processing steps are performed in Lines 16 to 20 to recover the embedding. The short-circuit edges are removed in Line 16, Lines 17 to 19 obtain a consistent orientation for all vertices (see Section 5.10), and Line 20 returns the combinatorial planar embedding of G in \tilde{G} along with an indication that G is planar. Theorem 6.3 asserts that this algorithm produces a combinatorial planar embedding in linear time.

Theorem 6.3 *The Planarity algorithm in Figure 14 (including the optimized data structures and subordinate routines defined in this paper) produces a combinatorial planar embedding \tilde{G} of a planar graph G in $O(n)$ time.*

Proof. The depth first search and lowpoint calculations in Line 1 are $O(n)$ [19]. The initialization in Line 2 is $O(n)$ by the methods described in Section 5. Lines 3 to 5 examine each edge of G and perform a constant time process for each tree edge to create a singleton biconnected component in \tilde{G} . Since the number of edges is $O(n)$, Lines 3 to 5 have an $O(n)$ cost in total. Line 6 performs n iterations. The loop on Line 7 examines each edge of a given vertex v to find back edges leading to descendants of v . This results in a total cost that is a constant factor of the number of edges, which is $O(n)$. Line 8 invokes the Walkup, which achieves constant amortized time by Lemma 6.1. Line 9 examines each edge of a vertex v to find tree edges leading to children of v , which again has an $O(n)$ total cost. Line 10 is a constant time operation per tree edge, or $O(n)$ in total. Line 11 invokes the Walkdown routine, which achieves constant amortized time by Lemma 6.2 for an $O(n)$ total cost. The loop on Line 12 again examines all edges of v , for a total $O(n)$ cost, and Line 13 executes a constant time operation on each back edge from v to a descendant, again resulting in an $O(n)$ total cost. Lines 14 and 15 are not applicable to planar graphs (but will also be shown to achieve $O(n)$ performance by Theorem 7.1). Line 16, the loop in Lines 17 and 18, and Line 19 are each simple $O(n)$ operations, and Line 20 is a constant time operation. \square

7 Kuratowski Subgraph Isolator

7.1 Determining the Non-planarity Minor

After performing a few preprocessing steps to delete the short-circuit edges and impart a consistent orientation to the vertices in each biconnected component of the partial embedding \tilde{G} , the process of isolating a Kuratowski subgraph begins by taking steps to select a non-planarity minor from those in Theorem 4.1 (see Figure 7) to be used as a basis for finding a $K_{3,3}$ or K_5 homeomorph. We first find an unembedded back edge (v, z) (i.e., a vertex z whose adjacentTo member still equals v at the end of step v), then we search the DFS

tree path from z until a vertex c is found whose `DFSParent` is v (thus, v^c is the root of a biconnected component on which the Walkdown failed). The first active vertices x and y along both external face paths emanating from v^c are then obtained (note that x and y are not necessarily neighbors of v^c because the short-circuit edges have been removed). If x or y is not a stopping vertex (i.e. if either has a non-empty `pertinentBicompList`), then the Walkdown on v^c must have been halted with a non-empty merge queue. The Walkdown can be called again to reconstruct the merge queue and obtain the last entry, which contains the biconnected component root r' whose non-virtual counterpart r is depicted in non-planarity minor A. Note that invoking the Walkdown again could result in more short-circuit edges, but these are deleted later because they are not used when isolating a $K_{3,3}$ homeomorph based on minor A. Alternately, one could simply postpone the short-circuit edge deletion until after the Walkdown invocation. Either way, the values of x and y must then be changed to the first active vertices both external face paths emanating from r' (again, note that x and y are not necessarily neighbors of r' once the short-circuit edges have been removed).

Next, a pertinent vertex w is obtained on the lower external face path between x and y (i.e. the external face path strictly between x and y that excludes the biconnected component root). If we have not already identified non-planarity minor A above, then non-planarity minor B can be used to isolate a Kuratowski subgraph if w has an externally active pertinent child biconnected component, which occurs when w has a non-empty `pertinentBicompList` in which the last element is a virtual vertex w^d , where the lowpoint of d is less than v . The failure of this test indicates that one of non-planarity minors C, D or E is applicable (according to Theorem 4.1), and each has an x - y path.

The highest x - y path is obtained by temporarily removing the internal edges incident to v^c , then traversing the proper face bordered by v^c and its two remaining incident edges. Due to the removal of the edges, the proper face may have a number of components separable by cut vertices. These are easily detected by the fact that the traversal arrives at vertices it has previously visited. The proper face traversal starts at v^c and moves toward x , pushing each visited vertex onto a stack. Each time a vertex on the external face path (v^c, \dots, x, \dots, w) is found, the stack is emptied the newly found vertex is pushed. Each time a previously visited vertex is encountered, the stack is popped up to the last occurrence of that vertex. As soon as a vertex along the external face path (v^c, \dots, y, \dots, w) is visited (and pushed), then the stack contains the list of vertices in the x - y path, with p_y on the top and p_x on the bottom. If either p_x or p_y is attached high (i.e. p_x is between v^c and x or p_y is between v^c and y), then non-planarity minor C can be used to isolate a Kuratowski subgraph.

If both p_x and p_y are attached low, then the internal edges of v^c are restored, and a test is made for non-planarity minor D by scanning the internal vertices of the x - y path for a vertex z whose x - y path edges are not consecutive (ignoring the possible intercession of the vertex structure for z). If an intervening edge exists, then it can be used by the proper face walk routine to traverse from z to v^c now that the internal edges of v^c have been restored. On the other hand, if the desired path for non-planarity minor D is not found, then Theorem 4.1 guarantees that non-planarity minor E is applicable, so an externally active vertex, possibly w but not p_x or p_y , exists along the lower external face path ($p_x, \dots, w, \dots, p_y$).

7.2 Marking a $K_{3,3}$ Homeomorph Based on Minors A, B, C and D

Since non-planarity minors A through D contain $K_{3,3}$, a subgraph homeomorphic to $K_{3,3}$ is isolated when one of those minors is found. Marking the edges and vertices associated with a Kuratowski subgraph consists mainly of traversing tree paths and external face cycles. A few unembedded back edges must also be added (and marked, including endpoints) to complete the Kuratowski subgraph.

In each minor, the dotted edge (u, x) may represent an unembedded back edge between an ancestor u of the current vertex v and a vertex x , or it may represent an unembedded back edge from u to a descendant of x plus the tree path from the descendant to x . The former case occurs if the leastAncestor member of x is less than v . In the latter case, u_x is the lowpoint of the first element c in the separatedDFSChildList of x , and the descendant d_x of x is the neighbor of u_x in G with the least DFI greater than or equal to c (such that d_x is in the DFS subtree rooted by c). If (u, x) represents a single unembedded back edge, then it is added and marked. Otherwise, the DFS tree path from d_x to x is marked by traversing the DFS tree path (for each vertex, the edge to the parent must be found so it can be marked), and the unembedded back edge (u_x, d_x) is added and marked.

The same routines apply to (u, y) and similar routines apply to (v, w) . If the adjacentTo member of w is set, then (v, w) represents a single unembedded edge. Otherwise, we obtain the last element w^c in the pertinentBicompList of w , then we obtain the descendant d_w of w by scanning the adjacency list of v in G to obtain the neighbor with the least DFI greater than or equal to c (such that d_w is in the DFS subtree rooted by c). If (v, w) represents a single unembedded back edge, then it is added and marked. Otherwise, the tree path from d_w to w is marked in the same way as the path from d_x to x , and the back edge from d_w to v is added and marked.

Given the above simple operations, marking the edges and vertices of a Kuratowski subgraph based on non-planarity minor A can be completed as follows. Traverse the external face of the biconnected component rooted by r' , marking all edges and vertices visited. Then, stopping vertices x and y and a pertinent vertex w can be found given the biconnected component root r' in the same manner as they are found for a biconnected component with root v^c in Section 7.1. Then, add the unembedded edges and mark paths corresponding to (u, x) , (u, y) and (v, w) as described above. Finally, mark the DFS tree path from r to whichever of u_x and u_y has the lower DFI.

Marking the edges and vertices of a Kuratowski subgraph based on non-planarity minor B is quite similar. Traverse the external face of the biconnected component rooted by v^c , marking all edges and vertices visited. Then, mark the edges and vertices corresponding to (u, x) , (u, y) as before. The last element w^z of the pertinentBicompList of w is obtained. Let u_z denote the lowpoint of z , let d_z denote the neighbor of u_z with the least DFI greater than or equal to z , and let d_w denote the neighbor of v with the least DFI greater than or equal to z . Mark the DFS tree paths from d_w to w , from d_z to z , and add and mark the edges (u_z, d_z) and (v, d_w) . Finally, mark the DFS tree path from $\max(u_x, u_y, u_z)$ to $\min(u_x, u_y, u_z)$. Although the endpoints v and $\max(u_x, u_y, u_z)$ are marked by other operations, the path from v to $\max(u_x, u_y, u_z)$ is not marked because the corresponding edge in non-planarity minor B, (u, v) , is not needed to form a $K_{3,3}$.

Non-planarity minor C represents two symmetric cases. If p_x is a high point of attachment, then the path corresponding to edge (v, y) is not required to form a $K_{3,3}$, so we remove the external face path from v^c to the nearer of y and p_y . On the other hand, if p_x is not

attached high, then p_y must be attached high, so instead we remove the path between v^c and x . Thus, if both p_x and p_y are attached high, then the path $(v^c, \dots, x, \dots, w, \dots, y, \dots, p_y)$ is marked. If only p_x is attached high, then the path $(v^c, \dots, x, \dots, w, \dots, y)$. If only p_y is attached high and the path $(x, \dots, w, \dots, y, \dots, v^c)$ is marked. The implementation is otherwise straightforward. The edges and vertices corresponding to (u, x) , (u, y) and (v, w) are marked as described above, the DFS tree path from v to the minimum of u_x and u_y is marked, and the x - y path, found by the process described in Section 7.1, is marked.

In non-planarity minor D, the edges (v, x) and (v, y) are not required to form a $K_{3,3}$. Marking the edges and vertices of a Kuratowski subgraph based on non-planarity minor D proceeds as follows. Mark the lower external face path (x, \dots, w, \dots, y) . Mark the x - y path and second internal path, which are found by the process described in Section 7.1. Finally, the edges and vertices corresponding to (u, x) , (u, y) and (v, w) are marked as described above, and the DFS tree path from v to the minimum of u_x and u_y is marked.

7.3 Marking a $K_{3,3}$ or K_5 Homeomorph Based on Minor E

Non-planarity minor E represents a K_5 minor, so the techniques to be used are quite similar to the prior isolators. However, before a Kuratowski subgraph can be isolated based on minor E, some additional cases must be considered since a K_5 minor often corresponds to a $K_{3,3}$ homeomorph rather than a K_5 homeomorph. Figure 15 depicts four additional $K_{3,3}$ minors. Minor E_1 occurs if the pertinent vertex w is not externally active (i.e. a second vertex z is externally active along the lower external face path strictly between p_x and p_y). If this condition fails, then $w = z$. Minor E_2 occurs if the external activity connection from w to an ancestor of v , denoted u_w , is a descendant of u_x and u_y . Minor E_3 occurs if u_x and u_y are distinct and at least one is a descendant of u_w . If none of these conditions occur, then we select minor E_4 if either $p_x \neq x$ or $p_y \neq y$. Finally, if none of the conditions for minors E_1 to E_4 occur, then a K_5 homeomorph can be obtained based on minor E.

As with minors A to D, there are symmetries to contend with and some edges of each minor are not needed to form a $K_{3,3}$. For minors E_1 and E_2 it is easy to handle the symmetries because, with a few assignments, they can be reduced to minors C and A, respectively. Minor E_3 does not require the edges (x, w) and (y, v) to form a $K_{3,3}$, and minor E_4 does not require the edges (u, v) and (w, y) to form a $K_{3,3}$. Moreover, note that the omission of edges from the external face of the biconnected component rooted by v must account for the fact that p_x or p_y may have been edge contracted into x or y in the depiction of the minor. For example, eliminating the edge (w, y) in minor E_4 corresponds to eliminating the path between w and p_y but not the path from p_y to y .

As for symmetries, minor E_1 in Figure 15(a) depicts z between x and w along the path $(x, \dots, z, \dots, w, \dots, y)$, but there is a symmetric case in which z appears between w and y along the path $(x, \dots, w, \dots, z, \dots, y)$. Also, Figure 15(c) depicts minor E_3 with u_x an ancestor of u_y , but there is a symmetric case in which u_y is an ancestor of u_x . For minor E_4 , Figure 15(d) depicts p_x distinct from x (and p_y can be equal to or distinct from y), but if $p_x = x$, then p_y must be distinct from y . Finally, note that the symmetric cases have different edges that have to be deleted to form a $K_{3,3}$.

Marking the edges and vertices of a Kuratowski subgraph based on non-planarity minor E_1 proceeds as follows. If the externally active vertex z is between p_x and w (as depicted in Figure 15(a)), then reduce to minor C after setting x equal to z such that p_x is a high point

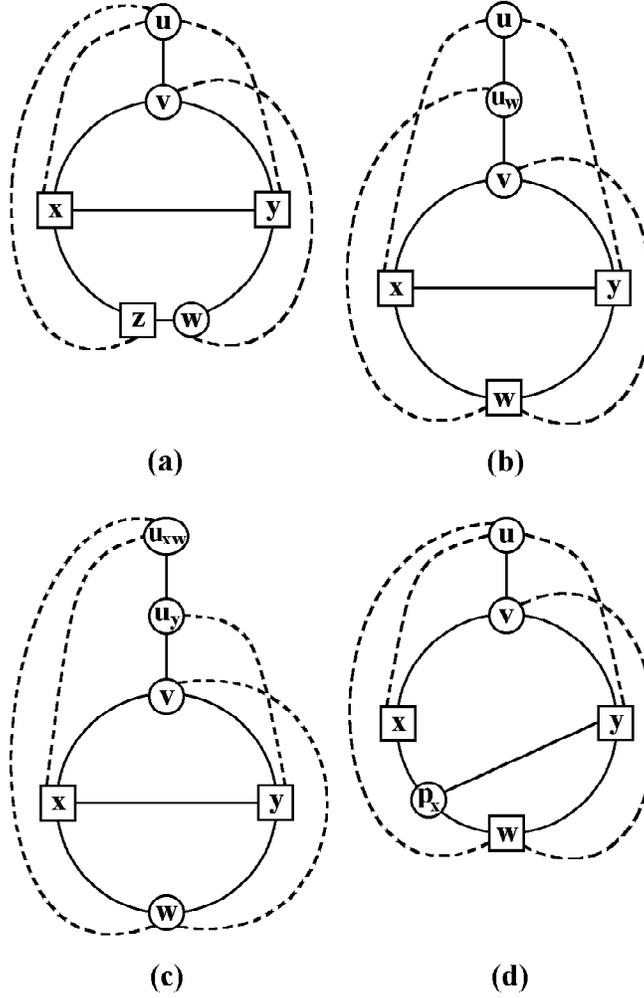


Figure 15: More $K_{3,3}$ Minors from Minor E. (a) Minor E_1 , (b) Minor E_2 , (c) Minor E_3 , (d) Minor E_4

of attachment for the x - y path. For the symmetric case in which z is between w and p_y , then reduce to minor C after setting y equal to z such that p_y is a high point of attachment for the x - y path.

Marking a Kuratowski subgraph based on non-planarity minor E_2 proceeds as follows. If the x - y path was previously marked by the implementation, then remove the markings. If the leastAncestor of w is less than v , then let u_w denote the leastAncestor of w and let d_w be equal to w . Otherwise, let u_w denote the lowpoint of the first child c in the separatedDFSChildList of w , and let d_w denote the the neighbor of u_w in G with the least DFI greater than or equal to c (such that d_w is in the DFS subtree rooted by c). Set the step variable v to u_w . If d_w equals w , then set the adjacentTo flag of w ; otherwise, set the pertinentBicomplList of w equal to the list containing only c . As a result, w satisfies the definition of pertinence for step u_w . Moreover, x and y are still externally active because u_x and u_y are ancestors of u_w . Thus, a reduction to minor A can now be performed.

It is interesting to note that while the planarity algorithm halts at the end of step v due to an unembedded back edge e from v to w or a descendant of w , the back edge e is not embedded and marked by the minor E_2 isolator described above. An isolator for minor E_2

that includes e can be constructed, but the method given is simpler to implement and still results in the isolation of a $K_{3,3}$ homeomorph.

Marking a Kuratowski subgraph based on non-planarity minor E_3 proceeds as follows. If u_x is an ancestor of u_y (i.e., $u_x < u_y$), then mark the edges and vertices along the external face path (v', \dots, p_x) that excludes w , and mark the edges and vertices along the external face path (w, \dots, y) that excludes v' . For the symmetric case in which u_y is an ancestor of u_x , mark the edges and vertices along the external face path (p_y, \dots, v') that excludes w , and mark the edges and vertices along the external face path (x, \dots, w) that excludes v' . The remaining steps are common to both cases. Add the unembedded back edges and mark the paths corresponding to edges (u, x) , (u, y) and (v, w) using the techniques previously described. Then, obtain u_w and d_w in the same manner as described for minor E_2 . Add the edge (u_w, d_w) and mark the DFS tree path (d_w, \dots, w) . Finally, mark the edges and vertices along the DFS tree path from v to the minimum of u_w , u_x , and u_y .

Marking a Kuratowski subgraph based on non-planarity minor E_4 proceeds as follows. If $p_x \neq x$, then mark the external face path (v', \dots, x, \dots, w) , and mark the external face path (p_y, \dots, v') that excludes w . For the symmetric case in which p_x equals x but $p_y \neq y$, mark the external face path (v', \dots, y, \dots, w) and mark the external face path (v', \dots, p_x) that excludes w . The remaining steps are common to both cases. Add the unembedded back edges and mark the paths corresponding to edges (u, x) , (u, y) and (v, w) using the techniques previously described. Then, obtain u_w and d_w in the same manner as described for minor E_2 . Add the edge (u_w, d_w) and mark the DFS tree path (d_w, \dots, w) . Finally, mark the edges and vertices along the DFS tree path from the minimum of u_w , u_x , and u_y to the maximum of u_w , u_x , and u_y .

As mentioned above, if the conditions for minors E_1 to E_4 do not occur, then the edges and vertices of a K_5 homeomorph can be marked based on minor E as follows. Mark all edges and vertices along the external face cycle rooted by v' . Add unembedded the back edges and mark the paths corresponding to edges (u, x) , (u, y) and (v, w) using the techniques previously described. Then, obtain u_w and d_w in the same manner as described for minor E_2 . Add the edge (u_w, d_w) and mark the DFS tree path (d_w, \dots, w) . Finally, mark the edges and vertices along the DFS tree path from v to the minimum of u_w , u_x , and u_y .

7.4 Kuratowski Subgraph Isolation in Linear Time

The process of isolating a Kuratowski subgraph of the input graph G is performed on the embedding structure \tilde{G} once the Walkdown fails to embed a back edge in a step v of the back edge embedding loop in procedure Planarity (see Figure 14). The process begins by removing short-circuit edges and imparting a consistent orientation to all vertices of each biconnected component in \tilde{G} . Then, one of the non-planarity minors is chosen, and a subgraph of G homeomorphic to $K_{3,3}$ or K_5 is marked in \tilde{G} based on the selected non-planarity minor. Finally, the virtual vertices are merged with their non-virtual counterparts, and all unmarked vertices and edges are removed. Theorem 7.1 asserts that these actions constitute a linear time algorithm for Kuratowski subgraph isolation.

Theorem 7.1 *The Planarity algorithm isolates a Kuratowski subgraph in a non-planar graph G in $O(n)$ time.*

Proof. The planarity operations prior to the call of the Kuratowski subgraph isolator are $O(n)$ by Lemma 6.1 and Theorem 6.3. The preprocessing steps for Kuratowski subgraph

isolation to remove short-circuit edges and impart a consistent orientation to each vertex of each biconnected component are $O(n)$, and the post-processing steps to merge any remaining virtual vertices and remove unmarked edges and vertices are also $O(n)$.

The selection of a non-planarity minor involves an $O(n)$ invocation of Walkdown to determine whether the merge queue is empty. Also, simple $O(n)$ loops are used to find stopping vertices x and y and a pertinent vertex w . Once these operations are performed, the tests that select minors A and B are constant time. If neither is selected, then the internal edges of v^c are removed and a proper face walk performed to find an x - y path and determines whether minor C can be selected. If not, then the internal edges of v^c are restored, and a portion of the proper face walk is conducted again to search for a second internal path for non-planarity minor D. The proper face walks are $O(n)$, and v^c has fewer than n incident edges. If minors A through D are not selected, then minor E is verified by a simple $O(n)$ loop that finds an externally active vertex along the path $(p_x, \dots, w, \dots, p_y)$.

Constant time tests are used to distinguish symmetric cases among the non-planarity minors and to select which secondary non-planarity minor to use in the case of minor E. Within each isolator, there are a constant number of simple $O(n)$ operations that mark edges and vertices along DFS tree paths and the external face of a biconnected component. The only non-trivial parts of the process are finding the x - y path and the second internal path for non-planarity minor D and finding the specific paths that map to edges (u, x) , (u, y) , and (v, w) as well as edges (v, z) and (u, z) in non-planarity minor B. The internal paths are found in $O(n)$ time by the aforementioned strategy of temporary edge removal and proper face walking. The paths are easy to handle as DFS tree path traversals once the proper descendants are found, which is optimized by a strategy that exploits the DFS numbering to quickly find an edge from an ancestor to any vertex in a DFS subtree by performing constant work per edge of the ancestor, for a total $O(n)$ cost. \square

8 PC-tree Problems

Recently, a new planarity test was presented by Shih and Hsu [18]. They develop the notion of a PC-Tree (described below) as a simplification of a PQ-tree. The algorithm has a number of similarities with our new planarity algorithms. The starting PC-tree is the depth first search tree, which is then processed in a bottom up fashion to embed the back edges from the current vertex, denoted i for a PC-tree (v in our algorithms), to the descendants of i . Unfortunately, the results of Shih and Hsu stated in [18] contain several errors that negate the proof of correctness of the planarity test, the claim of linear time performance and the claim of a fully defined Kuratowski subgraph isolator.

For example, Figure 16(a) depicts the K_5 minor pattern appearing in Figure 6(c) of [2] (with a few cosmetic changes), and Figure 16(b) depicts the corresponding PC-tree. The darkened triangles, called i -subtrees, represent subtrees rooted at x and y that have yet to be connected to v by unembedded back edges. Likewise, the whitened triangles, called i^* -subtrees, represent subtrees rooted at r , x , and y that have yet to be connected to an ancestor of v by unembedded back edges. The node labeled C is representative of a biconnected component, and its neighbors represent the essential nodes along the external face of the biconnected component. In PC-tree parlance, the nodes x and y are terminal nodes (i.e. they have at least one i -subtree, one i^* -subtree, and no descendant with the same

property [18, p. 181]).

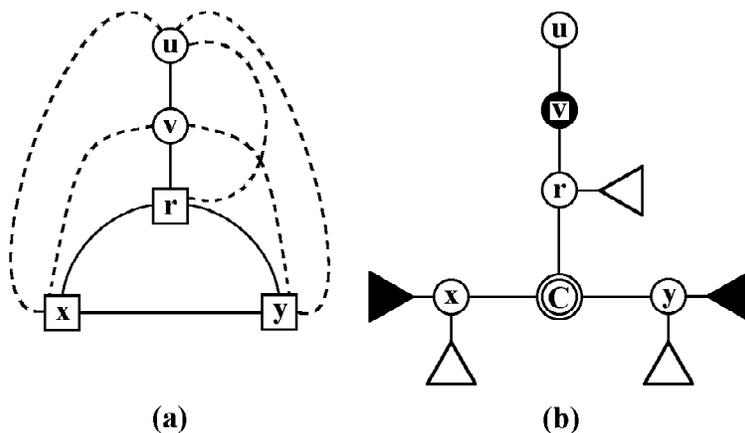


Figure 16: The K_5 non-planarity minor from [2], and the corresponding PC-tree at the beginning of step v

In [18], the only PC-tree pattern for detecting K_5 homeomorphs appears to only detect some K_5 minors. According to [18, p. 185] “we could have three terminal nodes being neighbors of a C-node, in which case we would get a subgraph homeomorphic to K_5 as illustrated in Fig. 6”. However, in Figure 16(b), node r is not a terminal node since x and y are its descendants. Thus, the C-node does not have three terminal nodes as neighbors in this case. Since [18] contains no other patterns for recognizing K_5 homeomorphs, one must conclude that an undefined non-planarity condition exists in the Shih-Hsu planarity test. In this case, an implementation would discover the non-planarity condition due to the test suggested by Lemma 2.5 in [18], but since the proof of that Lemma states that the result is a $K_{3,3}$ homeomorph, a Kuratowski subgraph isolator based on the results in [18] would clearly fail.

A more critical problem pertains to the correctness of the planarity testing algorithm itself. Corollary 2.8 and the prose on p. 187 of [18] clearly indicate how to continue processing under the configuration given in Figure 10 of [18], yet their planarity test should halt on this configuration if a descendant of i (the vertex being processed) that is also an ancestor of the C-node labeled W is connected by a back edge to an ancestor of vertex i (has an i^* -subtree). This condition arises in graphs such as the non-planarity minor in Figure 6(b) of [2], which is depicted (with a few cosmetic changes) in Figure 17(a). The dashed lines represent, at least in part, unembedded back edges. Our algorithm would currently be processing the back edges from v to its descendants. The corresponding PC-tree at the beginning of step v appears in Figure 17(b). In this example, x and y have i -subtrees, w and r have i^* -subtrees, and the C-node is a terminal node. Moreover, it is the only terminal node, and the work of Shih and Hsu [18] contains no theorem, lemma or corollary that detects a non-planarity condition in this configuration.

In essence, the non-planarity pattern depicted in Figure 17 is a counterexample to the proof of correctness appearing in Theorem 4.1 of [18]. Any reduction of the PC-tree in Figure 17(b) must retain r on the external face due to its i^* -subtree, which is infeasible since w must also be kept on the boundary cycle of a biconnected component due to its i^* -subtree. This problem negates the claim on p. 188 of Shih and Hsu [18] that the conditions established by “Lemma 2.5, Corollary 2.6 and Lemmas 3.1 and 3.2 ... imply a feasible internal embedding

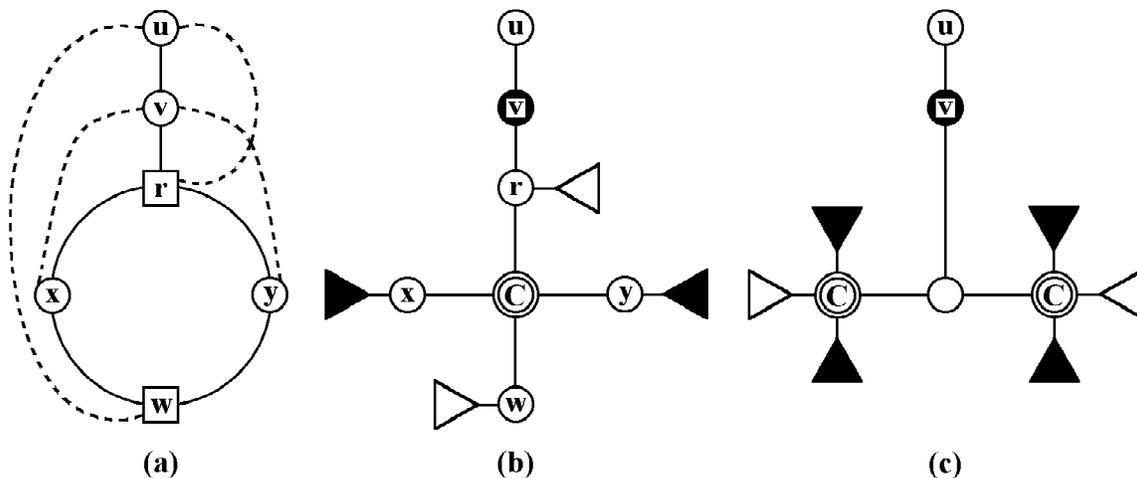


Figure 17: (a) A $K_{3,3}$ non-planarity minor from [2], (b) A corresponding PC-tree at the beginning of step v , (c) Another example of the PC-tree problem with two terminal nodes

for each biconnected component.” As such, Theorem 4.1 does not establish the Shih-Hsu algorithm as a correct planarity tester. In collaboration with graduate student Roland Wiese of the Eberhard-Karls University of Tübingen [21], this problem was determined to also be applicable when there are two terminal nodes, as depicted in Figure 17(c), because Lemma 3.1 does not apply to the terminal nodes. The problem appears to be an incorrect assumption in extending Lemma 2.7 and Corollary 2.8 to the case of having terminal nodes that are C-nodes, whose children cannot be arbitrarily permuted but must instead adhere to the order given by the representative bounding cycle (RBC).

It is easy to see that these cases have been missed in the processing model for PC-tree reduction, which is shown in Figure 11 of [18]. This figure shows how to reduce the PC-tree once it has been determined that a feasible internal embedding is possible according to the conditions established by Lemma 2.5, Corollary 2.6 and Lemmas 3.1 and 3.2. In Figure 11(i) and (ii), the first child of u in clockwise direction below the critical path P leads to an i^* -subtree. If it were changed to an i -subtree (a darkened subtree), then the resulting PC-tree is not reducible.

These additional non-planarity conditions and the non-planarity condition described by Lemma 3.1 of [18] also have an effect on the complexity analysis. In accounting for Lemma 3.1, p. 190 of [18] states that “determining which side of an intermediate C-node w contains i -subtrees (let v, v' be two neighbors of w in P) we only have to check the two neighbors of v (or v') in the cyclic list to see which one has the label i .” This may be true under the assumption that the input graph is planar, but the point of a planarity test (and of Lemma 3.1 in particular) is to determine *whether* the graph is planar, so some form of additional work must be done to determine whether the forbidden $i-i^*$ subtree pattern has occurred.

In terms of complexity analysis, there are additional concerns pertaining to the linear time performance of the algorithm as stated in [18]. For example, the claim that the “RBC will be stored as a circular doubly linked list” [18, p. 184] cannot be supported. When the representative bounding cycles of C-nodes must be joined together, the direction of traversal of two consecutive C-nodes may be reversed depending on which path contained the i -subtrees in each C-node. Joining the RBCs of two such C-nodes into a circular doubly linked list would require the inversion of links in the RBC nodes of one of the two C-nodes.

It is easy to create planar graphs in which $O(n^2)$ link inversions occur in total. It is therefore necessary to represent the RBC with a list that permits arbitrary link inversions as is done in the algorithms of this dissertation and as is depicted in Figure 1 of [2].

As a final concern, it appears that, in order to avoid $O(n^2)$ performance on PC-trees, one must not create PC-trees, at least not in the manner specified by [18, p. 184]: “we shall represent the 2-connected component by a C-node whose parent is $i \dots$ ”. Unfortunately, if every C-node (or P-node) indicated its actual parent in the PC-tree, then it is easy to create planar graphs on which $O(n^2)$ reparenting operations are performed on a set of C-nodes whenever their parent becomes part of a new C-node during a PC-tree reduction. Instead, it is necessary to let the parent of any P-node or C-node indicate a node in the RBC of its parent. The PC-tree parent of a node can be found by first following the parent link to some node in the RBC of the parent, then traversing the RBC until a node with a parent link is found. Thus, one is led repeatedly and inexorably to the methods of this paper and of [2] in order to create an algorithm that achieves linear time while exploiting those graph-theoretic properties that are common to both algorithms.

9 Conclusion and Related Solutions

This paper discussed the essential details of a new $O(n)$ planarity tester/embedder, as well as identifying some difficulties with the current formulation of PC-trees. A straightforward algorithm for isolating Kuratowski subgraphs was presented based on the non-planarity minors identified in the proof of correctness of the embedding algorithm. The first four non-planarity minors contain $K_{3,3}$, so marking the a $K_{3,3}$ homeomorph requires little more than traversal of external faces, tree paths and the addition of a few unembedded back edges. The unmarked vertices and edges are simply deleted. The fifth non-planarity minor is a K_5 minor, so four simple tests are performed to determine whether a $K_{3,3}$ or K_5 homeomorph can be isolated.

An $O(n)$ reference implementation of the new planarity algorithms described in this paper was created in three days for preliminary programming, four days for the planarity testing and embedding algorithms, and another three days for the Kuratowski subgraph isolator. The implementation was then tested on hundreds of millions of randomly generated graphs of up to 100 vertices plus well over a billion graphs generated with the aid of McKay’s nauty program [16] (specifically, all connected graphs with 11 or fewer vertices). For each graph, the integrity of the resulting combinatorial planar embedding or minimal non-planar subgraph was tested.

Extending this work to outerplanar graph embedding can literally be as simple as defining all vertices to be externally active at all times. A simplified version of our planarity proof of correctness yields $K_{2,3}$ and K_4 minors for outerplanarity obstruction isolation. Finally, little effort is required to modify the outerplanarity obstruction isolator so that it only finds $K_{2,3}$ homeomorphs. This is done by ignoring occurrences of K_4 except those in which a $K_{2,3}$ can also be found. Likewise, relatively little effort is required to modify the Kuratowski subgraph isolator so that it only finds $K_{3,3}$ homeomorphs. The result is $O(n)$ on graphs only containing a constant number of K_5 homeomorphs to ignore, and $O(n)$ in general if applied separately to the triconnected components of a graph. These algorithms will be presented in upcoming papers.

In terms of future work, we would like to reexamine the consecutive ones problem, which

was the original problem for which the PQ-tree was developed. A $(0, 1)$ -matrix has the *consecutive ones property for columns* if and only if its rows can be permuted so that in each column all of the ones are consecutive. It may be possible to create a reduction to the graph planarity problem based on a further examination of our data structures and detailed operations. Also, while graph drawing may often have application-specific requirements, a simple drawing method is often desirable as a starting point. An example is the *HorVert diagram*, which is a planar representation in which every vertex is represented by a horizontal line (or rectangle) and every edge is represented by a vertical line. While [10] presents an augmentation to the PQ-tree algorithm to create a HorVert diagram, their method is critically dependent on the single source aspect of the underlying st -numbering. We would also like to examine augmentations of our data structures that yield simplified methods for obtaining triconnected components as well as for enumerating, ranking and unranking planar embeddings. Finally, our algorithms may provide new insights that yield an $O(n)$ isolator for K_5 homeomorphs or a simplified $O(n)$ embedding algorithm for the projective plane.

Acknowledgements The authors are grateful to Paulette Lieby for providing valuable feedback on the first draft of this paper based on her implementation experience with these algorithms for the Magma Computational Algebra System (produced by the Computational Algebra Group within the School of Mathematics and Statistics of the University of Sydney). Roland Wiese of the Eberhard-Karls University of Tübingen also provided valuable commentary on the problems with the PC-tree formulation.

References

- [1] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and Systems Sciences*, 13:335–379, 1976.
- [2] J. Boyer and W. Myrvold. Stop minding your P’s and Q’s: A simplified $O(n)$ planar embedding algorithm. *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 140–146, 1999.
- [3] E. R. Canfield and S. G. Williamson. The two basic linear time planarity algorithms: Are they the same? *Linear and Multilinear Algebra*, 26:243–265, 1990.
- [4] N. Chiba, T. Nishizeki, A. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and Systems Sciences*, 30:54–76, 1985.
- [5] H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5(2):127–135, 1985.
- [6] N. Deo. Note on Hopcroft and Tarjan planarity algorithm. *Journal of the Association for Computing Machinery*, 23:74–75, 1976.
- [7] S. Even and R. E. Tarjan. Computing an st -numbering. *Theoretical Computer Science*, 2:339–344, 1976.
- [8] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

- [9] J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the Association for Computing Machinery*, 21(4):549–568, 1974.
- [10] R. Jayakumar, K. Thulasiraman, and M. N. S. Swamy. Planar embedding: Linear-time algorithms for vertex placement and edge ordering. *IEEE Transactions on Circuits and Systems*, 35(3):334–344, 1988.
- [11] M. Jünger, S. Leipert, and P. Mutzel. Pitfalls of using PQ-trees in automatic graph drawing. In G. Di Battista, editor, *Proc. 5th International Symposium on Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 193–204. Springer Verlag, Sept. 1997.
- [12] A. Karabeg. Classification and detection of obstructions to planarity. *Linear and Multilinear Algebra*, 26:15–38, 1990.
- [13] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.
- [14] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 215–232, New York, 1967. (Proc. Int. Symp. Rome, July 1966), Gordon and Breach.
- [15] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal of Computing*, 9(3):615–627, 1980.
- [16] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [17] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.
- [18] W.-K. Shih and W.-L. Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.
- [19] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
- [20] K. Wagner. Über einer eigenschaft der ebener complexe. *Math. Ann.*, 14:570–590, 1937.
- [21] R. Wiese. *Personal Communication*. June 6, 2001.
- [22] S. G. Williamson. Embedding graphs in the plane- algorithmic aspects. *Ann. Disc. Math.*, 6:349–384, 1980.
- [23] S. G. Williamson. Depth-first search and Kuratowski subgraphs. *Journal of the Association for Computing Machinery*, 31(4):681–693, 1984.
- [24] S. G. Williamson. *Combinatorics for Computer Science*. Computer Science Press, Rockville, Maryland, 1985.