Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 8, no. 3, pp. 241–273 (2004)

On the Cutting Edge: Simplified $O(n)$ Planarity by Edge Addition

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

Wendy J. Myrvold
University of Victoria
www.cs.uvic.ca/~wendym
wendym@cs.UVic.ca

Abstract

We present new $O(n)$-time methods for planar embedding and Kuratowski subgraph isolation that were inspired by the Booth-Lueker PQ-tree implementation of the Lempel-Even-Cederbaum vertex addition method. In this paper, we improve upon our conference proceedings formulation and upon the Shih-Hsu PC-tree, both of which perform comprehensive tests of planarity conditions embedding the edges from a vertex to its descendants in a 'batch' vertex addition operation. These tests are simpler than but analogous to the templating scheme of the PQ-tree. Instead, we take the edge to be the fundamental unit of addition to the partial embedding *while preserving planarity.* This eliminates the batch planarity condition testing in favor of a few localized decisions of a path traversal process, and it exploits the fact that subgraphs can become biconnected by adding a single edge. Our method is presented using only graph constructs, but our definition of external activity, path traversal process and theoretical analysis of correctness can be applied to optimize the PC-tree as well.

Article Type	Communicated by	Submitted	Revised
regular paper	P. Eades	September 2003	October 2004

The work by Wendy J. Myrvold was supported by NSERC.

1 Introduction

A *graph* G contains a set V of vertices and a set E of edges, each of which corresponds to a pair of vertices from V. Throughout this paper, n denotes the number of vertices of a graph and m indicates the number of edges. A *planar drawing* of a graph is a rendition of the graph on a plane with the vertices at distinct locations and no edge intersections (except at their common vertex endpoints). A graph is *planar* if it admits a planar drawing, and a *planar embedding* is an equivalence class of planar drawings described by the clockwise order of the neighbors of each vertex [9]. In this paper, we focus on a new $O(n)$-time planar embedding method. Generating a planar drawing is often viewed as a separate problem, in part because drawing algorithms tend to create a planar embedding as a first step and in part because drawing can be application-dependent. For example, the suitability of a graph rendition may depend on whether the graph represents an electronic circuit or a hypertext web site.

We assume the reader is familiar with basic graph theory appearing for example in [11, 26], including depth first search (DFS), the adjacency list representation of graphs, and the rationale for focusing on undirected graphs with no loops or parallel edges. We assume knowledge of basic planarity definitions, such as those for *proper face*, *external face*, *cut vertex* and *biconnected component*. We assume the reader knows that the input graph can be restricted to $3n - 5$ edges, enough for all planar graphs and to find a minimal subgraph obstructing planarity in any non-planar graph.

Kuratowski proved that a non-planar graph must contain a subgraph *homeomorphic* to either K_5 or $K_{3,3}$ (subgraphs in the form of K_5 (Figure 1(a)) or $K_{3,3}$ (Figure 1(b)) except that paths can appear in place of the edges). Just as a planar embedding provides a simple certificate to verify a graph's planarity, the indication of a *Kuratowski subgraph* (a subgraph homeomorphic to K_5 or $K_{3,3}$) provides a simple certificate of non-planarity. In some applications, finding a Kuratowski subgraph is a first step in eliminating crossing edges in the graph. For example, in a graph representing an integrated circuit, an edge intersection would indicate a short-circuit that could be repaired by replacing the crossed edge with a subcircuit of exclusive-or gates [17].

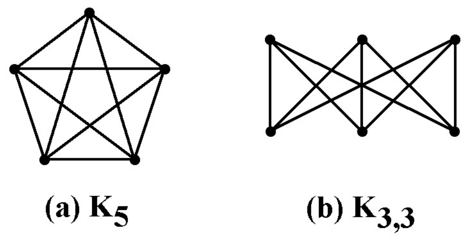

Figure 1: The planar obstructions K_5 and $K_{3,3}$.

The first $O(n)$-time planarity test algorithm is due to Hopcroft and Tarjan [12]. The method first embeds a cycle C of a biconnected 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 C. Since a path is added to the partial embedding when it is determined that planarity can be maintained, this method has sometimes been called the *path addition* method. The method is known to be complex (e.g. [6, p. 55]), though there are additional sources of information on this algorithm [8, 19, 21, 27, 28]. Its implementation in LEDA is slower than LEDA implementations of many other $O(n)$-time planarity algorithms [5].

The other well-known method of planarity testing proven to achieve linear time began with an $O(n^2)$-time algorithm due to Lempel, Even and Cederbaum [15]. The algorithm begins by creating an s,t-numbering for a biconnected graph. One property of an s,t-numbering is that there is a path of higher numbered vertices leading from every vertex to the final 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 algorithm was optimized to linear time by a pair of contributions. Even and Tarjan [10] 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. Since the permutations and flips of a PQ-tree are performed to determine whether a vertex can be added to the partial embedding while maintaining planarity, this method has become known as a *vertex addition* method. Chiba, Nishizeki, Abe and Ozawa [6] developed PQ-tree augmentations that construct a planar embedding as the PQ-tree operations are performed. Achieving linear time with the vertex addition method is also quite complex [14], partly because many PQ-tree templates are left for the reader to derive [1, p. 362]. There are also non-trivial rules to restrict processing to a 'pertinent' subtree of the PQ-tree, prune the pertinent subtree, and increase the efficiency of selecting and applying templates (more than one is often applied during the processing for one vertex).

An inspired planarity characterization by de Fraysseix and Rosenstiehl [7] leads to $O(n)$-time planarity algorithms, though the paper does not develop the linear time methodology. However, the planarity characterization is particularly interesting because it is the first to examine planarity in terms of conflicts between back edges as seen from a bottom-up view of the depth first search tree.

The planarity algorithm by Shih and Hsu [22, 23] was the first vertex addition method to examine the planarity problem using a bottom-up view of the depth first search tree. This represented a significant simplification over the PQ-tree. The method is based on the PC-tree [23], a data structure replacement for the PQ-tree which eliminates s,t-numbering, replaces Q-nodes with C-nodes, and detects PC-tree reducibility by testing a number of planarity conditions on P-nodes and C-nodes rather than the more complex PQ-tree templates.

The PC-tree method performs a post-order processing of the vertices according to their depth first order. For each vertex v, a small set of paths are identified as being pertinent to the planarity reduction in step v. The planarity conditions are tested along these paths; if all the conditions hold, then the PC-tree is reducible. The reduction has the effect of embedding all edges from v to its DFS descendants while maintaining planarity. To be sure that the reduction maintains planarity, a correct PC-tree algorithm must test the planarity conditions in [23] as well as the additional conditions identified in [3].

In 1996, the authors independently discovered that the PQ-tree could be eliminated from vertex addition planarity testing by exploiting the relationship between cut vertices, biconnected components and depth first search that were originally presented by Tarjan [24]. Using only graph constructs (i.e. attributes associated with vertices and edges of a graph data structure), we presented a vertex addition planarity algorithm in [2]. First, the vertices and depth first search (DFS) tree edges are placed into a partial embedding. Then, each vertex v is processed in reverse order of the depth first indices. The back edges from v to its DFS descendants are added to the partial embedding if it is determined that they can all be added while preserving planarity in the partial embedding. Our graph theoretic constructs for managing cut vertices and biconnected components are similar to the P-nodes and C-nodes of a PC-tree. Our planarity conditions are similar, though not identical due mainly to differences in low-level definitions and procedures. A more complete exposition of our vertex addition algorithm appears in [5], which also includes experimental results for a LEDA implementation that show it to be very fast in practice.

The expressed purpose of both PC-tree planarity and our own vertex addition method work was to present a simplified linear time planarity algorithm relative to the early achievements of Hopcroft and Tarjan [12] and Booth and Lueker [1]. While it is often accepted that the newer algorithms in [2] and [23] are simpler, the corresponding expositions in [5] and [3] clearly show that there is still complexity in the details. This additional complexity results from a 'batch' approach of vertex addition, in which back edges from a vertex to its descendants are embedded only after a set of planarity condition tests have been performed. These tests are simpler than but analogous to the templating scheme of the PQ-tree (another vertex addition approach).

Instead, we take the edge to be the fundamental unit of addition to the partial embedding *while preserving planarity* (just as vertex and path addition add a vertex or path while preserving planarity). Our new *edge addition* method exploits the fact that subgraphs can become biconnected by adding a single edge, eliminating the batch planarity condition testing of the prior vertex addition approaches in favor of a few localized decisions of a path traversal process. Non-planarity is detected at the end of step v if a back edge from v to a descendants was not embedded. Our edge addition method is presented using only graph constructs, but our graph theoretic analysis of correctness is applicable to the underlying graph represented by a PC-tree. Thus, our proof of correctness justifies the application of our definitions, path traversal process and edge embedding technique to substantially redesign the PC-tree processing, eliminating the numerous planarity conditions identified in [3, 23].

Section 2 describes the partial embedding data structure as a collection of the biconnected components that develop as edges are added, and it presents the fundamental operation of adding an edge to the partial embedding. Since an edge may biconnect previously separated biconnected components, they are merged together so that a single biconnected component results from adding the new edge. Section 3 explains the key constraint on the fundamental operation of adding an edge, which is that the external face must contain all vertices that will be involved in further edge additions. Due to this constraint, some biconnected components may need to be flipped before they are merged, and Section 4 describes how our method flips a biconnected component in constant time by relaxing the consistency of vertex orientation in the partial embedding.

Based on these principles, Section 5 presents our planar embedder, Section 6 proves that our planarity algorithm achieves linear time performance, and Section 7 proves that it correctly distinguishes between planar and non-planar graphs. The proof lays the foundation for our new Kuratowski subgraph isolator, which appears in Section 8. Finally, Section 9 presents concluding remarks.

2 The Fundamental Operation: Edge Addition

The planarity algorithm described in this paper adds each edge of the input graph G to an embedding data structure $\tilde{G}$ that maintains the set of biconnected components that develop as each edge is added. As each new edge is embedded in $\tilde{G}$, it is possible that two or more biconnected components will be merged together to form a single, larger biconnected component. Figure 2 illustrates the graph theoretic basis for this strategy. In Figure 2(a), we see a connected graph that contains a cut vertex r whose removal, along with its incident edges, separates the graph into the two connected components shown in Figure 2(b). Thus, the graph in Figure 2(a) is represented in $\tilde{G}$ as the two biconnected components shown in Figure 2(c). Observe that the cut vertex r is represented in each biconnected component that contains it. Observe that the addition of a single edge (v, w) with endpoints in the two biconnected components results in the single biconnected component depicted in Figure 2(d). Since r is no longer a cut vertex, only one vertex is needed in $\tilde{G}$ to represent it.

Indeed, Figure 2(d) illustrates the fundamental operation of the edge addition planarity algorithm. A single edge biconnects previously separable biconnected components, so these are merged together when the edge is embedded, resulting in a single larger biconnected component B. Moreover, the key constraint on this edge addition operation is that any vertex in B must remain on the external face of B if it must be involved in the future embedding of an edge. Hence, a biconnected component may need to be flipped before it is merged. For example, the lower biconnected component in Figure 2(d) was merged but also flipped on the vertical axis from r to w to keep y on the external face.

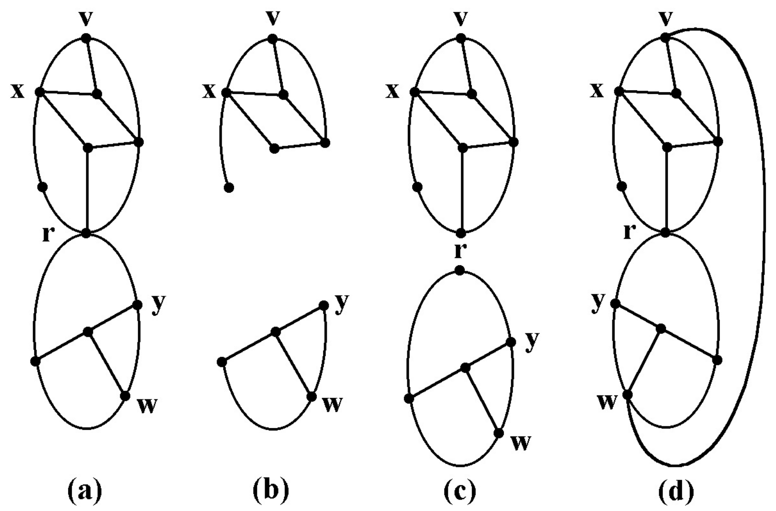

Figure 2: (a) A cut vertex r. (b) Removing r results in more connected components. (c) The biconnected components separable by r. (d) When edge (v, w) is added, r is no longer a cut vertex (by flipping the lower biconnected component, y remains on the external face).

3 External Activity

One of the key problems to be solved in an efficient planarity algorithm is how to add some portion of the input graph to the embedding in such a way that little or no further adjustment to the embedding is necessary to continue adding more of the input graph to the embedding. As described in Section 1, the PQ-tree method exploits a property of an s, t-numbered graph that allows it to create 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 $\tilde{G}_k$. We observe that an analogous property exists for a depth first search tree: Each vertex has a path of lower numbered DFS ancestors that lead to the DFS tree root. Therefore, our method processes the vertices in reverse order of their depth first indices (DFI) so that every unprocessed vertex has a path of unprocessed DFS ancestors leading to the last vertex to be processed, the DFS tree root. As a result, all unprocessed vertices must appear in a single face of the partial embedding $\tilde{G}$, and the flipping operations described in the prior section are designed to allow that face to be the common external face shared by all biconnected components in $\tilde{G}$.

As described in Section 2, the fundamental operation of adding a back edge may require merging of biconnected components, and some of those may need to be flipped so that vertices with unembedded edge connections to unprocessed vertices remain on the external face. Let w denote a DFS descendant of v in a biconnected component B. We say that w is *externally active* if there is a path from w to a DFS ancestor u of v consisting of a back edge plus zero or more DFS descendants of w, none of which are in B. Thus, an externally active vertex w will be involved in the future embedding of edges after the processing of v, either as the descendant endpoint of a back edge or as a cut vertex that will be a biconnected component merge point during the embedding of a back edge to some descendant of w. Figure 3 illustrates these cases.

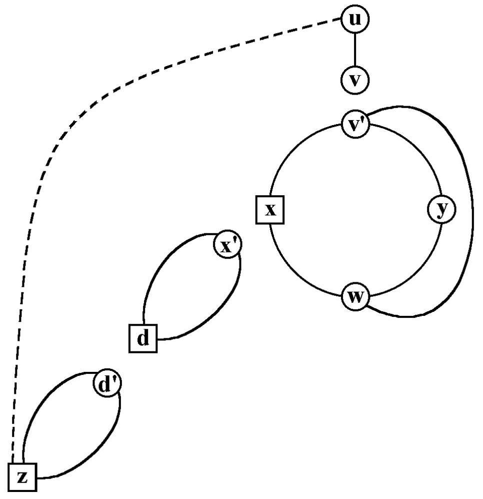

Figure 3: Externally active vertices are shown as squares and must remain on the external faces of the biconnected component that contain them. Vertex z is externally active because it is directly adjacent (in the input graph) to an ancestor u of v. Both x and d are externally active because they have a descendant *in a separate biconnected component* that is externally active.

The external activity of each vertex can be efficiently maintained as follows. The *lowpoint* of a vertex is the DFS ancestor of least DFI that can be reached by a path of zero or more descendant DFS tree edges plus one back edge, and it can be computed in linear time by a post-order traversal of the depth first search tree [24]. During preprocessing, we first obtain the *least ancestor* directly adjacent to each vertex by a back edge, then we compute the *lowpoint* of each vertex. Next, we equip each vertex with a list called *separatedDFSChildList*, which initially contains references to all DFS children of the vertex, sorted by their lowpoint values. To do this in linear time, categorize the vertices into ascending order by their lowpoint values, then add each to the end of the separatedDFSChildList of its DFS parent. To facilitate constant time deletion from a separatedDFSChildList, it is made circular and doubly linked, and each vertex is also equipped with a parameter that indicates the representative node for the vertex in the separatedDFSChildList of its DFS parent. When a biconnected component containing a cut vertex w and one of its children c is merged with the biconnected component containing w and the DFS parent of w, then the representative of c is deleted from the separatedDFSChildList of w. As a result, the separatedDFSChildList of w still contains references the children of w that remain in separate biconnected components from w. Thus, a vertex w is externally active during the processing of v if w either has a least ancestor less than v or if the first element in the separatedDFSChildList of w has a lowpoint less than v.

4 Flipping in Constant Time

Section 2 described the fundamental operation of adding a back edge to $\tilde{G}$ that might biconnect previously separated biconnected components in $\tilde{G}$. This necessitated merging the biconnected components together as part of adding the new edge to $\tilde{G}$. Then, Section 3 discussed the key constraint on the biconnected component merging process, which was that externally active vertices had to remain on the external face of the new biconnected component formed from the new edge and the previously separated biconnected components. This necessitated flipping some of the biconnected components.

The easiest method for flipping a biconnected component is to simply invert the adjacency list order, or *orientation*, of each of its vertices. However, it is easy to create graphs in which $\Omega(n)$ vertices would be inverted $\Omega(n)$ times during the embedding process. To solve this problem, we first observe that a biconnected component B in which vertex r has the least depth first index (DFI) never contains more than one DFS child of r (otherwise, B would not be biconnected since depth first search could find no path between the children except through r). We say that vertex r is the *root* of B, and the DFS tree edge connecting the root of B to its only DFS child c in B is called the *root edge* of B. In a biconnected component with root edge (r, c), we represent r with a *virtual vertex* denoted r^c to distinguish it from all other copies of r in $\tilde{G}$ (and r' denotes a root whose child is unspecified). Next, we observe that there are $O(n)$ biconnected component merge operations since each biconnected component root r' is only merged once and is associated with one DFS child of the non-virtual vertex r. Thirdly, we observe that a flip operation can only occur immediately before a merge. Therefore, a strategy that achieves constant time performance per flip operation will cost linear time in total. Finally, we observe that it is only a little more difficult to traverse the external face of a biconnected component if some vertices have a clockwise orientation and others have a counterclockwise orientation. Therefore, we flip a biconnected component by inverting the orientation of its root vertex.

A planar embedding with a consistent vertex orientation can be recovered in post-processing if an additional piece of information is maintained during embedding. We equip each edge with a *sign* initialized to $+1$. When a biconnected component must be flipped, we only invert the adjacency list orientation of the root vertex r^c so that it matches the orientation of the vertex r with which it will be merged. Then, the sign of the root edge (r^c, c) is changed to -1 to signify that all vertices in the DFS subtree rooted by c now have an inverse orientation. Since all of the edges processed by this operation were incident to a root vertex beforehand and a non-root vertex afterward, only constant work per edge is performed by all merge and flip operations during the embedding process. Moreover, a planar embedding for any biconnected component can be recovered at any time by imposing the orientation of the biconnected component root vertex on all vertices in the biconnected component. If the product of the signs along the tree path from a vertex to the biconnected component root vertex is -1, then the adjacency list of the vertex should be inverted. This is done with a cost commensurate with the biconnected component size by using a depth first search over the existing tree edges in the biconnected component.

Figure 4 helps to illustrate the biconnected component flip operation. Each vertex has a black dot and a white dot that signify the two pointers to the edges that attach the vertex to the external face (if it is on the external face). Observe the dots of each vertex to see changes of vertex orientation. Vertices 2 and 6 are square to signify that they are externally active due to unembedded back edges to vertex 0. The goal is to embed the edge $(1, 4)$.

In Figure 4(a), consistently following the black dot pointers yields a counterclockwise traversal of the external face of any biconnected component. In Figure 4(b), the biconnected component with root 3^4 is flipped so that edge $(1, 4)$ can be embedded along the left side while the externally active vertices 2 and 6 remain on the external face. Note that the orientations of vertices 4, 5 and 6 did not actually change relative to Figure 4(a). For example, the black dot in vertex 5 still leads to vertex 4, and the white dot in vertex 5 still leads to vertex 6. The result of embedding edge $(1, 4)$ appears in Figure 4(c).

Finally, we observe that the only new problem introduced by this technique is that extra care must be taken to properly traverse the external face of a biconnected component. In Figure 4(c), it is clear that following the black dots consistently no longer corresponds to a counterclockwise traversal of the external face of the biconnected component rooted by 1^2. The black and white dots signify links from a vertex to nodes of its adjacency list. A typical adjacency list representation has only one link or pointer from a vertex to a node its its adjacency list. In $\tilde{G}$, we use two link pointers so that we may indicate the nodes representing both edges that hold a vertex on the external face (if indeed the vertex is on the external face). These links can be traversed to obtain the edge leading to the next neighbor along the external face. However, the order of the links to the external face edges is reflective of the orientation of a vertex, and the orientation of vertices in a biconnected component can vary between clockwise and counterclockwise. Hence, whenever our method traverses an external face, it keeps track of the vertex w being visited but also the link to the edge that was used to enter w, and the opposing link is used to obtain the edge leading to the successor of w on the external face.

5 Planar Embedding by Edge Addition

Based on the key operations and definitions of the prior sections, this section presents our planar embedding algorithm. In the prior sections, we equipped the

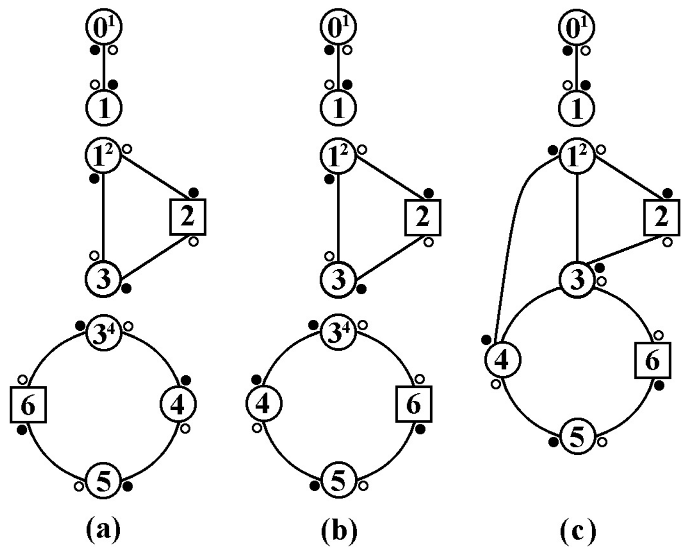

Figure 4: (a) $\tilde{G}$ before embedding edge $(1, 4)$. Vertices 2 and 6 are square to indicate they are externally active. (b) The biconnected component rooted by 3^4 is flipped, without altering the orientations of vertices 4 to 6, whose black and white dots indicate the same edges as they did before the flip. (c) Edge $(1, 4)$ is embedded, and vertices 2 and 6 have remained on the external face. The orientations of vertices 4 to 6 are not consistent with those of vertices 1 to 3, but the external face can still be traversed by exiting each vertex using whichever edge was not used to enter it.

vertices and edges of the embedding structure $\tilde{G}$ with some simple parameters and lists. In this section, only a few more additions are made as needed, and the full definition of $\tilde{G}$ appears in Appendix A.

The input graph G need not be biconnected nor even connected. The embedding algorithm begins with preprocessing steps to identify the depth-first search tree edges and back edges, to compute the least ancestor and lowpoint values, and to initialize the embedding structure $\tilde{G}$ so that it can maintain the notion of external activity as defined in Section 3. Then, each vertex v is processed in reverse DFI order to add the edges from v to its descendants. Figure 5 presents pseudocode for our planarity algorithm.

The DFS tree edges from v to its children are added first to $\tilde{G}$, resulting in one biconnected component consisting solely of the edge (v^c, c) for each child c of v. Although not necessary in an implementation, each biconnected component containing only a tree edge can be regarded as if it contained two parallel edges between the root vertex v^c and the child c so that the external face forms a cycle, as is the case for all larger biconnected components in $\tilde{G}$.

Next, the back edges from v to its descendants must be added. It is easy to find graphs on which $\Omega(n^2)$ performance would result if we simply performed

```
Procedure: Planarity
input: Simple undirected graph $G$ with $n \geq 2$ vertices and $m \leq 3n - 5$ edges
output: PLANAR and an embedding in $\tilde{G}$, or
        NONPLANAR and a Kuratowski subgraph of $G$ in $\tilde{G}$

(1) Perform depth first search and lowpoint calculations for $G$
(2) Create and initialize $\tilde{G}$ based on $G$, including creation of
        separatedDFSChildList for each vertex, sorted by child lowpoint

(3) For each vertex $v$ from $n - 1$ down to 0

(4)       for each DFS child $c$ of $v$ in $G$
(5)           Embed tree edge $(v^c, c)$ as a biconnected component in $\tilde{G}$

(6)       for each back edge of $G$ incident to $v$ and a descendant $w$
(7)           Walkup($\tilde{G}$, $v$, $w$)

(8)       for each DFS child $c$ of $v$ in $G$
(9)           Walkdown($\tilde{G}$, $v^c$)

(10)      for each back edge of $G$ incident to $v$ and a descendant $w$
(11)          if $(v^c, w) \notin \tilde{G}$
(12)              IsolateKuratowskiSubgraph($\tilde{G}$, $G$, $v$)
(13)              return (NONPLANAR, $\tilde{G}$)

(14) RecoverPlanarEmbedding($\tilde{G}$)
(15) return (PLANAR, $\tilde{G}$)
```

Figure 5: The edge addition planarity algorithm.

a depth first search of the DFS subtree rooted by v to find all descendant endpoints of back edges to v. Therefore, we must restrict processing to only the *pertinent subgraph*, which is the set of biconnected components in $\tilde{G}$ that will be merged together due to the addition of new back edges to v. Our method identifies the pertinent subgraph with the aid of a routine we call the Walkup, which is discussed in Section 5.1. Once the pertinent subgraph is identified, our method adds the back edges from v to its descendants in the pertinent subgraph, while merging and flipping biconnected components as necessary to maintain planarity in $\tilde{G}$ and keeping all externally active vertices on the external faces of the biconnected components embedded in $\tilde{G}$. This phase is performed with the aid of a routine we call the Walkdown, which is discussed in Section 5.2.

Embedding a tree edge cannot fail, and the proof of correctness in Section 7 shows that the Walkdown only fails to embed all back edges from v to its descendants if the input graph is non-planar. When this occurs, a routine discussed in Section 8 is invoked to isolate a Kuratowski subgraph. Otherwise, if all tree edges and back edges are embedded in every step, then the planar embedding is recovered as discussed in Section 4.

The remainder of this section discusses the Walkup and Walkdown. To understand their processing, we make a few more definitions, mostly to set the notion of a pertinent subgraph into the context of our embedding structure $\tilde{G}$, which manages a collection of biconnected components. A biconnected component with root w^c is a *child biconnected component* of w. A non-virtual (hence non-root) vertex w descendant to the current vertex v is *pertinent* if G has a back edge (v, w) not in $\tilde{G}$ or w has a child biconnected component in $\tilde{G}$ that contains a pertinent vertex. A *pertinent biconnected component* contains a pertinent vertex. An *externally active biconnected component* contains an externally active vertex (our definition of an externally active vertex, given above, applies only to non-virtual vertices). Vertices and biconnected components are *internally active* if they are pertinent but not externally active, and vertices and biconnected components are *inactive* if they are neither internally nor externally active.

5.1 The Walkup

In this section, we discuss a subroutine called Walkup. Pseudo-code for the Walkup appears in Appendix C. As mentioned above, the Walkup is invoked by the core planarity algorithm once for each back edge (v, w) to help identify the pertinent subgraph. This information is consumed by the Walkdown, which embeds back edges from v to its descendants in the pertinent subgraph.

Specifically, the purpose of the Walkup is to identify vertices and biconnected components that are pertinent due to the given back edge (v, w). Vertex w is pertinent because it is the direct descendant endpoint of a back edge to be embedded. Each cut vertex in $\tilde{G}$ along the DFS tree path strictly between v and w, denoted $T_{v,w}$, is pertinent because the cut vertex will become a merge point during the embedding of a back edge.

To help us mark as pertinent the descendant endpoints of back edges to v, we equip each vertex with a flag called *backedgeFlag*, which is initially cleared during preprocessing. The first action of the Walkup for the back edge (v, w) is to raise the backedgeFlag flag of w. Later, when the Walkdown embeds the back edge (v, w), the backedgeFlag flag is cleared. To help us mark as pertinent the cut vertices in $\tilde{G}$ along the DFS tree path $T_{v,w}$ between v and w, we equip the vertices with an initially empty list called *pertinentRoots*. Let r be a cut vertex in $\tilde{G}$ along $T_{v,w}$ with a DFS child s also in $T_{v,w}$. Then, r^s is a biconnected component root along $T_{v,w}$ in $\tilde{G}$. When the Walkup finds r^s, it adds r^s to the pertinentRoots of r. Later, when the Walkdown merges r and r^s, then r^s is removed from the pertinentRoots list of r. Thus, a vertex is determined to be pertinent if its backedgeFlag flag is set or if its pertinentRoots list is non-empty.

Although not necessary, the Walkdown benefits if the Walkup places the roots of externally active biconnected components after the roots of internally active biconnected components. Thus, a biconnected component root r^s is prepended to the pertinentRoots list of r unless r^s is the root of an externally active biconnected component (i.e. if lowpoint$(s) < v$), in which case it is appended.

A few strategies must be employed to efficiently implement the traversal phase of the Walkup that sets the pertinentRoots lists of cut vertices in $\tilde{G}$ along $T_{v,w}$. The goal is to ensure that the work done by all invocations of Walkup in step v is commensurate with the sum of the sizes of the proper faces that will be formed when the back edges from v to its descendants are added to $\tilde{G}$. Therefore, we do not simply traverse the tree path $T_{v,w}$ in $\tilde{G}$ looking for biconnected component roots. Instead, our method traverses the external face of each biconnected component that contains part of $T_{v,w}$ to find the root. If w is the point of entry in a biconnected component, there are two possible external face paths that can be traversed to obtain the root r^s. Since the path must become part of the bounding cycle of a proper face, it cannot contain an externally active vertex between w and r^s. However, it is not known which path is longer, nor if either contains an externally active vertex. Therefore, both paths are traversed in parallel to ensure that r^s is found with a cost no greater than twice the length of the shorter external face path. Once r^s is located, it is added to the pertinentRoots of r, then r is treated as the entry point of the biconnected component, and the Walkup reiterates until $r^s = v^c$.

By traversing external face paths in parallel, rather than simply searching the tree path $T_{v,w}$, a Walkup can find the roots of biconnected components made pertinent by the back edge (v, w). However, the intersection of T_{v,w_1} and T_{v,w_2} for two back edges (v, w_1) and (v, w_2) can be arbitrarily large, so the Walkup must be able to detect when it has already identified the roots of pertinent biconnected components along a path due to a prior invocation for another back edge of v. To solve this problem, we equip each vertex with a flag called *visited*, which is initially cleared. Before visiting any vertex, the Walkup checks whether the visited flag is set, and ends the invocation if so. Otherwise, after visiting the vertex, the Walkup sets the visited flag (all visited flags that were set are cleared at the end of each step v, an operation that can be done implicitly if visited is an integer that is set when equal to v and clear otherwise). Figure 6 illustrates the techniques described here. Appendix C contains a complete pseudo-code description of this operation.

5.2 The Walkdown

In this section, we discuss the Walkdown subroutine (see Appendix D for the pseudo-code). As mentioned above, the Walkdown is invoked by the core planarity algorithm once for each DFS child c of the vertex v to embed the back edges from v to descendants of c. The Walkdown receives the root v^c of a biconnected component B. The Walkdown descends from v^c, traversing along the external face paths of biconnected components in the pertinent subgraph (identified by the Walkup described in Section 5.1). When the Walkdown encounters the descendant endpoint w of a back edge to v, the biconnected components visited since the last back edge embedding are merged into B, and the edge (v^c, w) is then added to B. As described in Section 4, each biconnected components

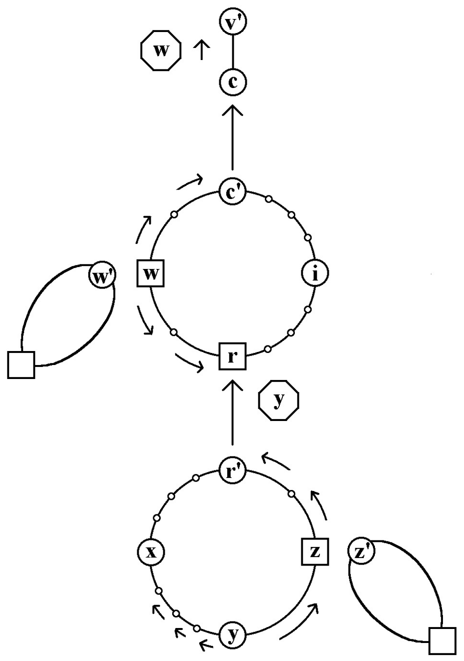

Figure 6: In this example, the Walkup is invoked first for back edge (v, w) then for (v, y). The first Walkup sets the backedgeFlag for w, then it proceeds from w in both directions around the external face, setting the visited flag for each vertex. The clockwise traversal finds the biconnected component root first, at which point c' is recorded in the pertinentRoots of c, and the Walkup begins simultaneous traversal again at c and terminates at v'. The second Walkup sets the backedgeFlag of y, then begins the simultaneous traversal phase. This time the counterclockwise traversal finds the biconnected component root first after passing around an externally active vertex (only the Walkdown embeds edges, so only the Walkdown must avoid passing over an externally active vertex). Once r' is recorded in the pertinentRoots of r, the Walkup resets for simultaneously traversal at r and is immediately terminated because the first Walkup set the visited flag of r (so any ancestor biconnected component roots have already been recorded in the pertinentRoots members of their non-root counterparts).

that is merged into B may be flipped if necessary to ensure that externally active vertices remain on the external face of B. The external face paths traversed by the Walkdown become part of the new proper face formed by adding (v^c, w).

To embed the back edges, the Walkdown performs two traversals of the external face of B, corresponding to the two opposing external face paths emanating from v^c. The traversals perform the same operations and are terminated by the same types of conditions, so the method of traversal will only be described once.

A traversal begins at v^c and proceeds in a given direction from vertex to vertex along the external face in search of the descendant endpoints of a back edges. Whenever a vertex is found to have a pertinent child biconnected component, the Walkdown descends to its root and proceeds with the search. Once the descendant endpoint w of a back edge is found, the biconnected component roots visited along the way must be merged (and the biconnected components flipped as necessary) before the back edge (v^c, w) is embedded. An initially empty *merge stack* is used to help keep track of the biconnected component roots to which the Walkdown has descended as well as information that helps determine whether each biconnected component must be flipped when it is merged.

A biconnected component must be flipped if the direction of traversal upon entering a cut vertex r changes when the traversal exits the root r^s. For example, in Figure 4(a), v^c is 1^2, and the first traversal exits 1^2 on the black dot link and enters vertex 3 on the white dot link. The traversal then descends to the root 3^4, where it must exit using the white dot link to avoid the externally active vertex 6. Since entry and exit are inverse operations, opposite colored links must be used to avoid a flip. When the same link color is used for both, the direction of traversal has changed and a flip is required. Note that once the orientation of the root vertex 3^4 is inverted in Figure 4(b), the traversal exits from the black dot link to reach vertex 4. As the above example shows, in order to be able to merge and possibly flip a biconnected component, the following pieces of information must have been stored in the merge stack when a Walkdown traversal descended from a vertex r to the root of one of its pertinent child biconnected components: the identity of the root r^s (from which the non-root r is easily calculated), the direction of entry into r, and the direction of exit from r^s.

A Walkdown traversal terminates either when it returns to v^c or when it encounters a non-pertinent, externally active vertex, which we call a *stopping vertex*. If it were to proceed to embed an edge after passing a stopping vertex, then the stopping vertex would not remain on the external face, but it must because it is externally active. Note that a pertinent externally active vertex may become a stopping vertex once the Walkdown embeds the edges that made the vertex pertinent.

Prior to encountering a stopping vertex, if a vertex w is encountered that has more than one pertinent child biconnected component, then the Walkdown must descend to an internally active child biconnected component if one is available. Note that the Walkdown traverses the entire external face of an internally active child biconnected component and returns to w, but once the Walkdown descends to a pertinent externally active child biconnected component, it will encounter a stopping vertex before returning to w (traversal cannot proceed beyond the first externally active vertex in the child biconnected component). So, to ensure that all back edges that can be embedded while preserving planarity are in fact embedded, the Walkdown enforces Rule 1.

Rule 1 *When vertex w is encountered, first embed a back edge to w (if needed) and descend to all of its internally active child biconnected components (if any) before processing its pertinent externally active child biconnected components.*

If the Walkup uses the strategy from Section 5.1 of always putting externally active child biconnected component roots at the end of the pertinentRoots lists, then the Walkdown can process the internally active child biconnected components first by always picking the first element of a pertinentRoots list.

A similar argument to the one above also governs how the Walkdown chooses a direction from which to exit a biconnected component root r^s to which it has descended. Both external face paths emanating from r^s are searched to find the first internally or externally active vertices x and y in each direction (i.e. inactive vertices are skipped). The path along which traversal continues is then determined by Rule 2.

Rule 2 *When selecting an external face path from the root r^s of a biconnected component to the next vertex, preferentially select the external face path to an internally active vertex if one exists, and select an external face path to a pertinent vertex otherwise.*

Finally, if both external face paths from r^s lead to non-pertinent externally active vertices, then both are stopping vertices and the entire Walkdown (not just the current traversal) can be immediately terminated due to a non-planarity condition described in the proof of correctness of Section 7. Figure 7 provides another example of Walkdown processing that corresponds to the example of Walkup in Figure 6.

With two exceptions, the operations described above allow the total cost of all Walkdown operations to be linear. Traversal from vertex to vertex and queries of external activity and pertinence are constant time operations. Merge and flip operations on biconnected components cost constant time per edge. The data structure updates that maintain external activity and pertinence are constant time additions to the biconnected component merge operation. Finally, adding an edge takes constant time. Thus, the cost of all Walkdown operations performed to add an edge can be associated with the size of the proper face that forms when the new edge is added, except that when Walkdown descends to a biconnected component root, it traverses both external face paths emanating from the root, but only one of those paths becomes part of the proper face formed when the new edge is added. It is possible to construct graphs in which a path of length $\Omega(n)$ is traversed but not selected $\Omega(n)$ times. A second problem is that the cost of traversing the path between the last embedded back edge endpoint and the stopping vertex is not associated with a proper face.

Both of these costs can be bounded to a linear total by the introduction into $\tilde{G}$ of *short-circuit edges*, which are specially marked edges added between v^c and the stopping vertex of each Walkdown traversal (except of course when the traversal is terminated by the above non-planarity condition). The short-circuit edge forms a new proper face, which removes the path from the last back edge endpoint to the stopping vertex from the external face. Moreover, this method reduces to $O(1)$ the cost of choosing the path to proceed along after desdending to a biconnected component root. In step v, the immediate external face neighbors of a pertinent root r^s are active because the interceding inactive

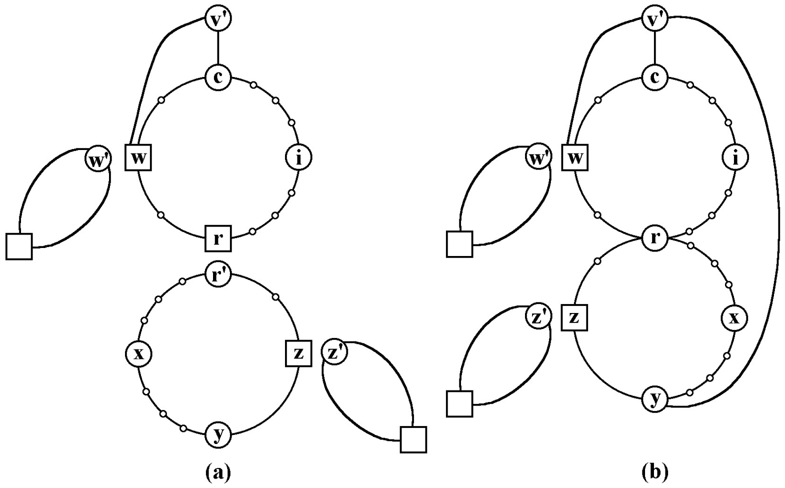

Figure 7: (a) The Walkdown embedded the edge (v, w). It descended from v' to c, and then from c to c'. The first active vertices along the external face paths emanating from c' are w and r. Arbitrarily, w is chosen. (b) Once (v, w) is added, w becomes a stopping vertex, so a second traversal begins at v' and traverses the external path through c, i, and down to r. Since r is pertinent, the Walkdown descends to r' and finds the active vertices. In part (a), the counterclockwise direction gives y and the clockwise direction gives z. Since z is not pertinent, the path toward y is selected, but the direction of traversal entering r was clockwise. Thus, the biconnected component rooted by r' must be flipped when r' is merged with r during the addition of back edge (v, y).

vertices are removed from the external face by short-circuit edges embedded in step r. Since only two short-circuit edges are added per biconnected component, and each can be associated with a unique vertex (the DFS child in its root edge), at most $2n$ short-circuit edges are added. Short-circuit edges are also specially marked, so they can be easily removed during the post-processing steps that recover a planar embedding or Kuratowski subgraph of G from $\tilde{G}$.

5.3 A More Global View

The proof of correctness in Section 7 shows the essential graph structures that occur when the Walkdown fails to embed a back edge, but an indication of the original pertinent subgraph is not essential to proving that the graph is non-planar when the Walkdown fails to embed an edge. Yet, it is instructive to see an example of the overall effect of a Walkdown on the entire pertinent subgraph. Figure 8 shows the state immediately before the Walkdown of an example set of biconnected components (ovals), externally active vertices (squares), and descendant endpoints of unembedded back edges (small circles). The dark ovals are internally active, the shaded ovals are pertinent but externally active, and the light ovals are non-pertinent. Figure 9 shows the result of the Walkdown processing over the example of Figure 8.

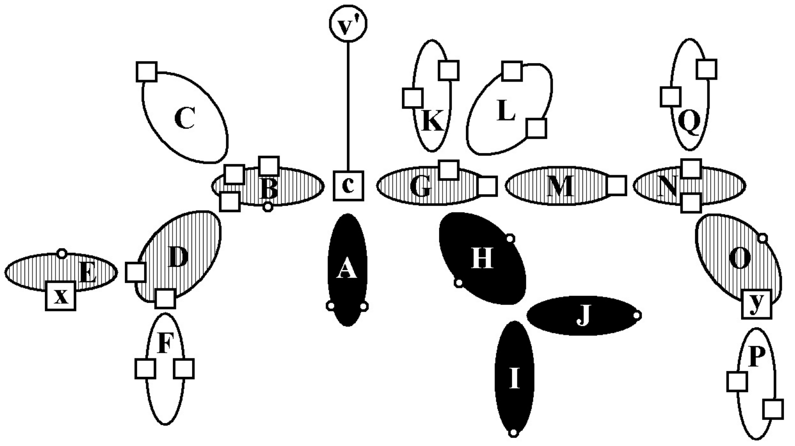

Figure 8: Before the Walkdown on v'.

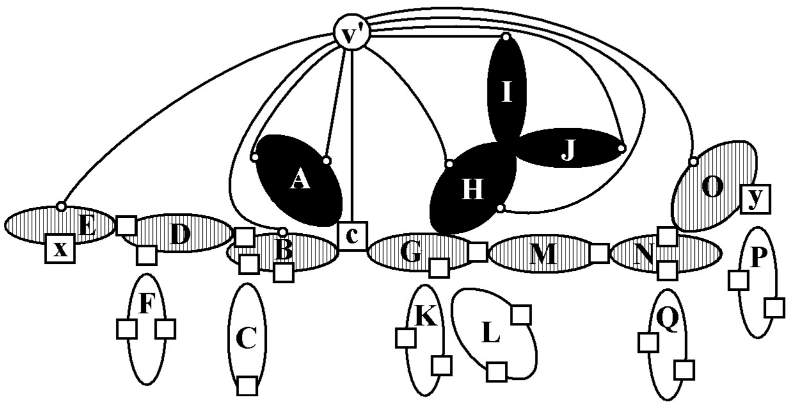

Figure 9: After the Walkdown on v'.

The first traversal Walkdown descends to vertex c, then biconnected component A is selected for traversal because it is internally active, whereas B and G are pertinent but externally active. The back edges to vertices along the external face of A are embedded and then the traversal returns to c. Biconnected component B is chosen next, and it is flipped so that traversal can proceed toward the internally active vertex in B. The back edge to the vertex in B is embedded, and the root of B is merged with c. Then, the traversal proceeds to the non-virtual counterpart of the root of D, which is externally active because D is externally active. The traversal continues to the root of D then to the non-virtual counterpart of the root of E rather than the non-virtual counterpart of the root of F; both are externally active, but the path to the former is selected because it is pertinent. Traversal proceeds to the internally active vertex in E to embed the back edge, at which time D and E become part of the biconnected component rooted by v'. Finally, traversal continues along E until the first traversal is halted by the stopping vertex x.

The second Walkdown traversal proceeds from v' to c to the biconnected component G, which is flipped so that the internal activity of H, I and J can be resolved by embedding back edges. The back edges to I and J are embedded between the first and second back edges that are embedded to H. The bounding cycles of the internally active biconnected components are completely traversed, and the traversal returns to G. Next, the roots of M, N and O are pushed onto the merge stack, and N is also flipped so that the traversed paths become part of the new proper face that is formed by embedding the back edge to the vertex in O. Finally, the second traversal is halted at the stopping vertex y.

Generally, the first traversal embeds the back edges to the left of tree edge (v', c), and the second traversal embeds the back edges on the right. As this occurs, the externally active parts of this graph are kept on the external face by permuting the children of c (i.e. selecting A before B and G) and by biconnected component rotations. The internally active biconnected components and pertinent vertices are moved closer to v' so that their pertinence can be resolved by embedding back edges. The internally active vertices and biconnected components become inactive once their pertinence is resolved, which allows them to be surrounded by other back edges as the Walkdown proceeds.

Overall, our algorithm proceeds directly from the simple task of identifying the pertinent subgraph directly to the edge embedding phase, which the vertex addition methods perform during the 'planarity reduction' phase. Essentially, we prove in Section 7 the sufficiency of augmenting the reduction phase with Rules 1 and 2, avoiding the intermediate phase in vertex addition methods that first tests the numerous planarity conditions identified in [3, 23] for the PC-tree, in [5] for our prior vertex addition method, and in [1] for the PQ-tree.

6 Linear Time Performance

In this section we consider the strategies used to make our planarity algorithm achieve $O(n)$ performance. The result is stated as Theorem 1.

Theorem 1 *Given a graph G with n vertices, algorithm* Planarity *determines whether G is planar in $O(n)$ time.*

Proof. The depth first search and least ancestor and lowpoint calculations are implemented with well-known linear time algorithms. Section 3 described a linear time method for initializing the data structures that maintain external activity, which consists of creating a separatedDFSChildList for each vertex, each sorted by the children's lowpoint values. During the run of the main edge embedding loop, the cost of embedding each tree edge is $O(1)$, resulting in linear time total cost. Section 5.1 describes a method for implementing the Walkup to identify the pertinent subgraph with a cost commensurate with the sizes of the proper faces that will be formed by back edges when they are embedded. Similarly, Section 5.2 describes a method for implementing the Walkdown to embed the back edges within the pertinent subgraph with a cost commensurate with the sizes of the proper faces that will be formed by back edges and short-circuit edges. Since the sum of the degrees of all proper faces is twice the number of edges, a total linear cost is associated with all Walkup and Walkdown operations. The cost of the loops that invoke Walkup and Walkdown are associated with the back edges and tree edges, respectively, for which the functions are invoked, yielding constant cost per edge. At the end of the main loop, the test to determine whether any back edge was not embedded by the Walkdown results in an additional constant cost per back edge, for a total linear cost. □

Corollary 2 *Given a planar graph G with n vertices, algorithm* Planarity *produces a planar embedding of G in $O(n)$ time.*

Proof. First, the edges of G are added to $\tilde{G}$ by the main loop of Planarity in $O(n)$ time by Theorem 1. The short-circuit edges added to optimize the Walkdown (see Section 5.2) are specially marked, so their removal takes $O(n)$ time. Then, the vertices in each biconnected component are given a consistent vertex orientation as described in Section 4. This operation can do no worse than invert the adjacency list of every non-root vertex for a constant time cost per edge, or $O(n)$ in total. Finally, $\tilde{G}$ is searched for any remaining biconnected component roots in $O(n)$ time (there will be more than one if G is not biconnected), and they are merged with their non-root counterparts (without flipping) for a cost commensurate with the sum of the degrees of the roots, which is $O(n)$. □

7 Proof of Correctness

In this section, we prove that the algorithm Planarity described in Section 5 correctly distinguishes between planar and non-planar graphs. It is clear that the algorithm maintains planarity of the biconnected components in $\tilde{G}$ as an invariant during the addition of each edge (see Corollary 4). Thus, a graph G is planar if all of its edges are added to $\tilde{G}$, and we focus on showing that if the algorithm fails to embed an edge, then the graph must be non-planar.

For each vertex v, the algorithm first adds to $\tilde{G}$ the tree edges between v and its children without the possibility of failure. Later, the back edges from v to its descendants are added by the routine called Walkdown described in Section 5.2. The main algorithm Planarity invokes the Walkdown once for each DFS child c of v, passing it the root v^c of a biconnected component B in which it must embed the back edges between v^c and descendants of c.

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 a pertinent child biconnected component root, a condition depicted in Figure 10(a). Otherwise, if the Walkdown halts on B without embedding all back edges from v^c to descendants of c, then each Walkdown traversal was terminated by a stopping vertex on the external face of B, a condition depicted by Figure 10(b).

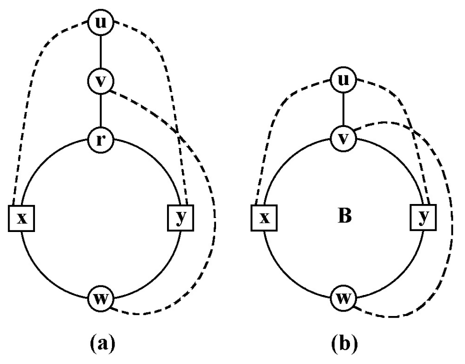

Figure 10: Walkdown halting configurations. Square vertices are externally active, all edges may be paths, and dashed edges include an unembedded back edge. All ancestors of the current vertex v are contracted into u. (a) The Walkdown has descended from the current vertex v to the root r of a biconnected component, but the pertinent vertex cannot be reached along either external face path due to stopping vertices x and y. (b) Each Walkdown traversal from v has encountered a stopping vertex in the biconnected component B that contains a root copy of v. The Walkdown could not reach the pertinent vertex w due to the stopping vertices. Although this configuration is planar, Theorem 3 proves the existence of additional edges that form one of four non-planar configurations.

In Figure 10(a), u represents the contraction of the unprocessed ancestors of v so that (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 root copy of r. Square vertices are externally active. 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 tree path containing a separated DFS child of the descendant and an unembedded back edge to the ancestor of v. Similarly, Figure 10(b) shows stopping vertices x and y that prevent traversal from reaching a pertinent vertex w in a biconnected component rooted by a root copy of v.

Both diagrams depict minors of the input graph. Since Figure 10(a) depicts a $K_{3,3}$, the input graph is non-planar. However, Figure 10(b) appears to be planar, so it is natural to ask why the Walkdown did not first embed (v, w) then embed (v, x) such that (v, w) is 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 11. Theorem 3 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.

Theorem 3 *Given a biconnected connected component B with root v^c, if the* Walkdown *fails to embed a back edge from v to a descendant of c, then the input graph G is not planar.*

Proof. By contradiction, suppose the input graph is planar but the Walkdown halts without embedding a back edge. To do so, the Walkdown must encounter a stopping vertex. If this occurs because stopping vertices were encountered along both external face paths emanating from the root of a pertinent child biconnected component, then the Walkdown terminates immediately, and the $K_{3,3}$ depicted in Figure 11(a) shows that the input graph is non-planar. Hence, the Walkdown must halt on stopping vertices on the external face of the biconnected component containing v^c.

Figure 11(b) results if the pertinent vertex w has an externally active pertinent child biconnected component. Embedding the connection from a separated descendant of w to v inside B would place an externally active vertex z inside B. Thus, the input graph is non-planar since Figure 11(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 $v, \ldots, x, \ldots, w$, and the other along the path $v, \ldots, y, \ldots, w$. 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.

In Figure 11(c), the x-y path has p_x attached closer to v than x. Note that p_y can also be attached closer to v than y. In fact, Figure 11(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 11(c) contains a $K_{3,3}$.

In Figure 11(d), 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 11(d) contains a $K_{3,3}$.

In Figure 11(e), an externally active vertex (possibly distinct from w) exists 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 11(e) represents a K_5 minor.

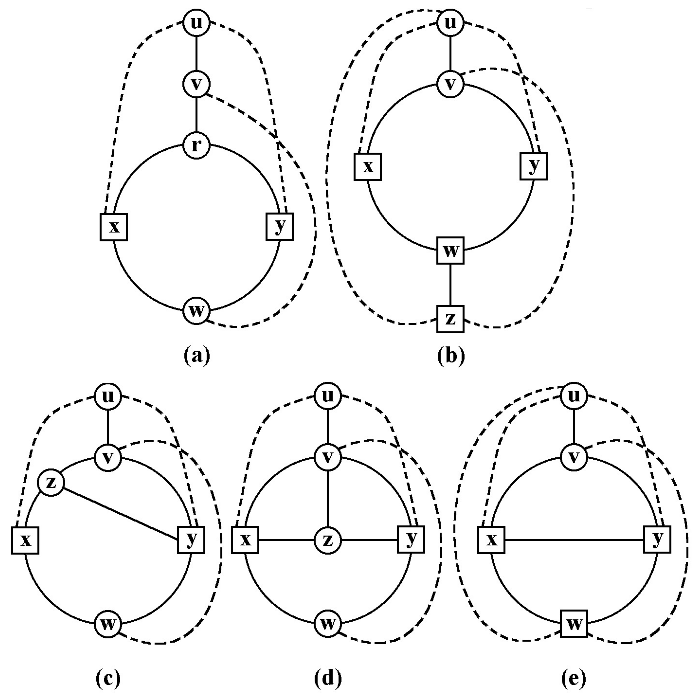

Figure 11: Non-planarity minors of the input graph.

Finally, suppose for the purpose of contradiction that the Walkdown failed to embed a back edge and none of the non-planarity conditions described above exist. As mentioned above, due to the absence of the condition of Figure 11(a), the two Walkdown traversals must have ended on stopping vertices along external face paths in the biconnected component B rooted by v^c. By the contradictive assumption, B has a pertinent vertex w along the lower external face path strictly between stopping vertices x and y. We address two cases based on whether or not there is an obstructing x-y path.

If no obstructing x-y path exists, then at the start of step v all paths between x and y in $\tilde{G}$ 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 and then processes the internally active child biconnected components first (see Rule 1), so the pertinence of w must be the result of an externally active pertinent child biconnected component. Yet, this contradicts the pertinence of w since the condition of Figure 11(b) does not exist.

On the other hand, suppose there is an obstructing x-y path, but non-planarity minors C to E do not 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^c were removed, along with any resulting separable components. 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 at the beginning of step v. Let r_1 denote whichever of p_x or p_y was the root of that biconnected component, and let r_2 denote one of p_x or p_y such that $r_1 \neq r_2$. Since the condition of Figure 11(c) does not exist, r_2 is equal to or an ancestor of x or y and was therefore externally active when the Walkdown descended to r_1^s (a root copy of r_1, where s is equal to or a DFS ancestor of r_2). Moreover, the first active vertex along the path that is now the highest x-y path was r_2 because the condition of Figure 11(d) does not exist. Descending from r_1^s 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 11(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 (see Rule 2). □

Corollary 4 *Algorithm* Planarity *determines whether a graph G is planar.*

Proof. For each vertex v in reverse DFI order, the edges from v to its descendants are embedded. The embedding of tree edges cannot fail, and if the Walkdown fails to embed a back edge from v to a descendant, then Theorem 3 shows that the graph is not planar. Hence, consider the case in which Planarity embeds all edges from v to its descendants in each step. The edges from v to its ancestors are therefore embedded as those ancestors are processed. When a tree edge is added to $\tilde{G}$, planarity is maintained since the tree edge is added into a biconnected component by itself. When a back edge is added, the preceding merge and flip of biconnected components maintain planarity, and the new back edge is added in the external face region incident to two vertices currently on the external face. Thus, planarity is maintained in $\tilde{G}$ for all edges added, so G is planar if all of its edges are added to $\tilde{G}$. □

8 Kuratowski Subgraph Isolator

The non-planarity minors of Figure 11 can be used to find a Kuratowski subgraph in a non-planar graph (or a subgraph with at most $3n-5$ edges). Because the method is closely associated with the back edges, external face paths and DFS tree paths of the input graph, the linear time performance and correctness of the method are clear from the discussion.

The first step is to determine which non-planarity minor to use. Minors A to D can be used directly to find a subgraph homeomorphic to $K_{3,3}$. Minor E is a K_5 minor, so a few further tests are performed afterward to determine whether a subgraph homeomorphic to $K_{3,3}$ or K_5 can be obtained. To determine the minor, we first find an unembedded back edge (v, d), then search up the DFS tree path $T_{v,d}$ in $\tilde{G}$ to find the root v^c of a biconnected component on which the Walkdown failed. The Walkdown can then be reinvoked to determine whether the merge stack is empty (unless the merge stack is still available from the

Walkdown that halted). Either way, the short-circuit edges should be deleted and $\tilde{G}$ should be properly oriented as described in Section 4 before proceeding.

If the merge stack is non-empty, then the desired biconnected component root r can be found at the top of the stack. Otherwise, we use v^c. The two external face paths from the selected root are searched for the stopping vertices x and y, then we search the lower external face path $(x, \ldots, y)$ for a pertinent vertex w that the Walkdown could not reach. Then, if the merge stack was non-empty, we invoke the minor A isolator (the isolators are described below).

If the merge stack is empty, then we must choose between minors B to E. If w has a pertinent externally active child biconnected component (check the last element of the pertinentRoots list), then we invoke the minor B isolator. Otherwise, we must find the highest x-y path by temporarily deleting the internal edges incident to v^c, then traversing the proper face bordered by v^c and its two remaining edges. Due to the removal of edges, the bounding cycle of the face will contain cut vertices, which can be easily recognized and eliminated as their cut vertices are visited for a second time during the walk. Once the x-y path is obtained, the internal edges incident to v^c are restored.

If either p_x or p_y is attached high, then we invoke the minor C isolator. Otherwise, we test 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 above the x-y path. If it exists, such a vertex z may be directly incident to v^c or it may have become a cut vertex during the x-y path test. Either way, we invoke the minor D isolator if z is found and the minor E isolator if not.

Each isolator marks the vertices and edges to be retained, then deletes unmarked edges and merges biconnected components. The edges are added and marked to complete the pertinent path from w to v and the external activity paths from x and y to ancestors of v. Minors B and E also require an additional edge to complete the external activity path for z. Finally, the tree path is added from v to the ancestor of least DFI associated with the external activity of x, y and (for minors B and E) z. Otherwise, we mark previously embedded edges along depth first search tree paths, the x-y path and v-z path, and the external face of the biconnected component containing the stopping vertices.

To exemplify marking an external activity path, we consider the one attached to x (in any of the non-planarity minors). If the least ancestor directly attached to x by a back edge (a value obtained during the lowpoint calculation) is less than v, then let u_x be that least ancestor, and let d_x be x. Otherwise, u_x is the lowpoint of the first child χ in the separatedDFSChildList of x, and let d_x be the neighbor of u_x in G with the least DFI greater than χ. We mark the DFS tree path from d_x to x and add and mark the edge (u_x, d_x). The external activity paths for y and, when needed, z are obtained in the same way.

Marking the pertinent path is similar, except that minor B requires the path to come from the pertinent child biconnected component containing z. In the other cases, the backedgeFlag flag tells whether we let d_w be w. If the backedgeFlag flag is clear or we have minor B, then we obtain the last element w^χ in the pertinentRoots list of w, then scan the adjacency list of v in G for the neighbor d_w with least DFI greater than χ. Finally, mark the DFS tree path d_w to w and add the edge (v, d_w).

To conclude the $K_{3,3}$ isolation for minor A, we mark the DFS tree path from v to the least of u_x and u_y and we mark the external face of the biconnected component rooted by r. For minor B, we mark the external face path of the biconnected component rooted by v^c and the DFS tree path from $\max(u_x, u_y, u_z)$ to $\min(u_x, u_y, u_z)$. The path from v to $\max(u_x, u_y, u_z)$, excluding endpoints, is not marked because the edge (u, v) in minor B is not needed to form a $K_{3,3}$. For the same reason, minors C and D omit parts of the external face of the biconnected component rooted by v^c, but both require the tree path v to $\min(u_x, u_y)$. Minor C omits the short path from p_x to v if p_x is attached high, and otherwise it omits the short path from p_y to v. Minor D omits the upper paths $(x, \ldots, v)$ and $(y, \ldots, v)$. In all cases, the endpoints of the omitted paths are not omitted.

Finally, the minor E isolator must decide between isolating a $K_{3,3}$ homeomorph and a K_5 homeomorph. Four simple tests are applied, the failure of which implies that minor E can be used to isolate a K_5 homeomorph based on the techniques described above. The first test to succeed implies the ability to apply the corresponding minor from Figure 12.

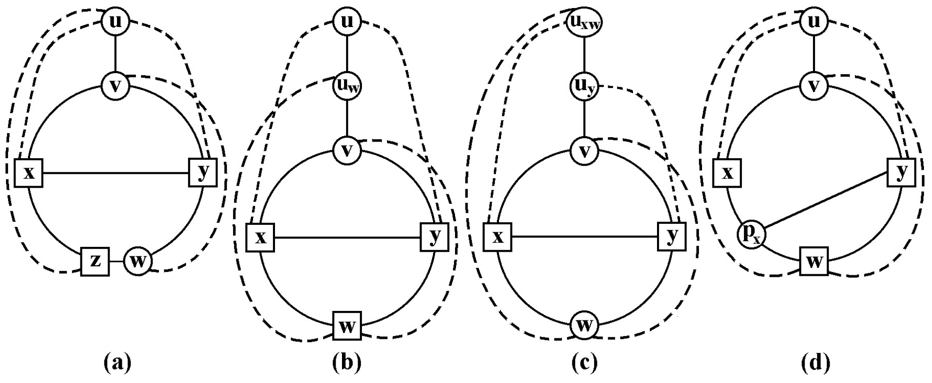

Figure 12: (a) Minor E_1. (b) Minor E_2. (c) Minor E_3. (d) Minor E_4.

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 u_w of v 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. Minor E_4 occurs if either $p_x \neq x$ or $p_y \neq y$.

As with minors A to D, there are symmetries to contend with and some edges that are not needed to form a $K_{3,3}$. For minors E_1 and E_2 it is easy to handle the symmetries because they reduce to minors C and A, respectively. Minor E_3 does not require (x, w) and (y, v) to form a $K_{3,3}$, and minor E_4 does not require (u, v) and (w, y) to form a $K_{3,3}$. Moreover, note that the omission of these edges 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 (e.g. eliminating (w, y) in minor E_4 corresponds to eliminating the path $(w, \ldots, p_y)$ but not $(p_y, \ldots, y)$).

As for symmetries, minor E_1 in Figure 12(a) depicts z between x and w along the path $(x, \ldots, z, \ldots, w, \ldots, y)$, but z may instead appear between w and y along the path $(x, \ldots, w, \ldots, z, \ldots, y)$. Also, Figure 12(c) depicts minor E_3 with u_x an ancestor of u_y, but u_y could instead be an ancestor of u_x. For minor E_4, Figure 12(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, the symmetric cases have different edges that have to be deleted to form a $K_{3,3}$.

9 Conclusion

This paper discussed the essential details of our new 'edge addition' planarity algorithm as well as a straightforward method for isolating Kuratowski subgraphs. These algorithms simplify linear time graph planarity relative to prior approaches. Our implementation has been rigorously tested on billions of randomly generated graphs and all graphs on 12 or fewer vertices (generated with McKay's nauty program [18]). Our implementation, as well as independent implementations such as [16, 25], have required only a few weeks to implement.

Our implementation of the edge addition method as well as a LEDA implementation of our earlier vertex addition formulation in [2] have both been found to be competitive with implementations of the well-known prior methods, including being several times faster than LEDA's Hopcroft-Tarjan and PQ-tree implementations [5]. Although some PC-tree implementations exist [13, 20], none that are suitable for empirical comparisons are currently available publicly. Yet the empirical results in [5] suggest that PC-tree planarity can be quite fast, with a similar performance profile to our own earlier vertex addition method (based on the similarities of the algorithms).

However, in [5], edge addition methods were found to be faster, and only our 'edge addition' implementation was found to be competitive with the Pigale implementation of an algorithm based on the planarity characterization by de Fraysseix and Rosenstiehl [7]. At the 11^{th} International Graph Drawing Symposium, Patrice Ossona de Mendez noted that some of the many optimizations applied to the underlying graph data structures of Pigale could be applied to further speed up our implementation. Even without these optimizations, our implementation was found to be just as fast with a Gnu compiler and about 35 percent faster with a Microsoft compiler [4, 5].

Future research shall include reporting extensions of our method to outerplanarity and three instances of the subgraph homeomorphism problem as well as investigation of a fourth subgraph homeomorphism problem, the consecutive ones problem and interval graph recognition, and the generation of maximal planar subgraphs, visibility representations, and alternate planar embeddings. Our methods may also assist in the development of simplified, efficient embedders for the projective plane and the torus.

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] J. M. Boyer. Additional PC-tree planarity conditions. In J. Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing 2004*, to appear in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[4] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. Technical Report RT-DIA-83-2003, Dipartimento di Informatica e Automazione, Universitá di Roma Tre, Nov. 2003.

[5] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In G. Liotta, editor, *Proceedings of the 11th International Symposium on Graph Drawing 2003*, volume 2912 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 2004.

[6] 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.

[7] H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5(2):127–135, 1985.

[8] N. Deo. Note on Hopcroft and Tarjan planarity algorithm. *Journal of the Association for Computing Machinery*, 23:74–75, 1976.

[9] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, Upper Saddle River, NJ, 1999.

[10] S. Even and R. E. Tarjan. Computing an *st*-numbering. *Theoretical Computer Science*, 2:339–344, 1976.

[11] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[12] J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the Association for Computing Machinery*, 21(4):549–568, 1974.

[13] W.-L. Hsu. An efficient implementation of the PC-trees algorithm of shih and hsu's planarity test. Technical Report TR-IIS-03-015, Institute of Information Science, Academia Sinica, July 2003.

[14] M. Jünger, S. Leipert, and P. Mutzel. Pitfalls of using PQ-trees in automatic graph drawing. In G. Di Battista, editor, *Proceedings of the 5th International Symposium on Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 193–204. Springer Verlag, Sept. 1997.

[15] 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. Gordon and Breach.

[16] P. Lieby. Planar graphs. *The Magma Computational Algebra System*, http://magma.maths.usyd.edu.au/magma/htmlhelp/text1185.htm.

[17] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal of Computing*, 9(3):615–627, 1980.

[18] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

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

[20] A. Noma. Análise experimental de algoritmos de planaridade. Master's thesis, Universidade de São Paulo, May 2003. in Portuguese.

[21] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[22] W.-K. Shih and W.-L. Hsu. A simple test for planar graphs. In *Proceedings of the International Workshop on Discrete Mathematics and Algorithms*, pages 110–122, 1993.

[23] W.-K. Shih and W.-L. Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.

[24] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[25] A.-M. Törsel. *An implementation of the Boyer-Myrvold algorithm for embedding planar graphs.* University of Applied Sciences Stralsund, Germany, 2003. Diploma thesis, in German.

[26] D. B. West. *Introduction to Graph Theory.* Prentice Hall, Inc., Upper Saddle River, NJ, 1996.

[27] S. G. Williamson. Embedding graphs in the plane- algorithmic aspects. *Annals of Discrete Mathematics*, 6:349–384, 1980.

[28] S. G. Williamson. *Combinatorics for Computer Science.* Computer Science Press, Rockville, Maryland, 1985.

A Graph Structure for Embedding

class **Graph**
- n: integer, number of vertices
- m: integer, number of edges
- V: array $[0 \ldots n-1]$ of Vertex
- R: array $[0 \ldots n-1]$ of RootVertex
- E: array $[0 \ldots 6n-1]$ of HalfEdge
- S: stack of integers, the merge stack

class **Vertex**
- link: array $[0 \ldots 1]$ of AdjacencyListLink
- DFSparent: integer
- leastAncestor: integer
- lowpoint: integer
- visited: integer
- backedgeFlag: integer
- pertinentRoots: list of integers
- separatedDFSChildList: list of integers
- pNodeInChildListOfParent: pointer into separatedDFSChildList of DFSParent

class **RootVertex**
- link: array $[0 \ldots 1]$ of AdjacencyListLink
- parent: integer, index into V

class **HalfEdge**
- link: array $[0 \ldots 1]$ of AdjacencyListLink
- neighbor: integer
- sign: 1 or -1

class **AdjacencyListLink**
- type: enumeration of {inV, inR, inE }
- index: integer, index into V, R or E

B Low-Level Operations

```
inactive($w$) ::= not pertinent($w$) and not externallyActive($w$)
internallyActive($w$) ::= pertinent($w$) and not externallyActive($w$)
pertinent($w$) ::= backedgeFlag of $w$ set or pertinentRoots of $w$ is non-empty

externallyActive($w$) ::=
    leastAncestor of $w$ less than $v$ or
    lowpoint of first member of $w$'s separatedDFSChildList is less than $v$

GetSuccessorOnExternalFace($w$, $w_{in}$)
    $e \leftarrow$ link[$w_{in}$] of $w$
    $s \leftarrow$ neighbor member of $e$
    if $w$ is degree 1, $s_{in} \leftarrow w_{in}$
    else $s_{in} \leftarrow$ (link[0] of $s$ indicates HalfEdge twin of $e$) ? 0 : 1
    return ($s$, $s_{in}$)

MergeBiconnectedComponent($S$) ::=
    Pop 4-tuple $(r, r_{in}, r^c, r^c_{out})$ from $S$
    if $r_{in} = r^c_{out}$,
        Invert orientation of $r^c$ (swap links in $r^c$ and throughout
            adjacency list)
        Set sign of $(r^c, c)$ to -1
        $r^c_{out} \leftarrow 1$ xor $r^c_{out}$

    for each HalfEdge $e$ in adjacency list of $r^c$
        Set neighbor of $e$'s twin HalfEdge to $r$

    Remove $r^c$ from pertinentRoots of $r$
    Use $c$'s pNodeInChildListOfParent to remove $c$ from $r$'s
        separatedDFSChildList

    Circular union of adjacency lists of $r$ and $r^c$ such that
        HalfEdges link[$r_{in}$] from $r$ and link[$r^c_{out}$] from $r^c$ are consecutive
        link[$r_{in}$] in $r \leftarrow$ link[1 xor $r^c_{out}$] from $r^c$
```

C Walkup Pseudo-Code

Procedure: Walkup
this: Embedding Structure $\tilde{G}$
in: A vertex w (a descendant of the current vertex v being processed)

```
(1) Set the backedgeFlag 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 root vertex, $z' \leftarrow x$
(8)         else if $y$ is a root 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 pertinentRoots of $z$
(16)              else Prepend $z'$ to the pertinentRoots of $z$
(17)           $(x,\, x_{in}) \leftarrow (z,\, 1)$
(18)           $(y,\, y_{in}) \leftarrow (z,\, 0)$

(19)        else $(x, x_{in}) \leftarrow$ GetSuccessorOnExternalFace($x,\, x_{in}$)
(20)             $(y, y_{in}) \leftarrow$ GetSuccessorOnExternalFace($y,\, y_{in}$)
```

D Walkdown Pseudo-Code

Procedure: Walkdown
this: Embedding Structure $\tilde{G}$
in: A root vertex v' associated with DFS child c

```
(1) Clear the merge stack $S$

(2) for $v'_{out}$ in $\{0, 1\}$
(3)      $(w, w_{in}) \leftarrow$ GetSuccessorOnExternalFace($v'$, 1 xor $v'_{out}$)
(4)      while $w \neq v'$
(5)           if the backedgeFlag member of $w$ is equal to $v$,
(6)              while stack $S$ is not empty,
(7)                   MergeBiconnectedComponent($S$)
(8)              EmbedBackEdge($v'$, $v'_{out}$, $w$, $w_{in}$)
(9)              Clear the backedgeFlag member of $w$ (assign $n$)

(10)          if the pertinentRoots of $w$ is non-empty,
(11)             Push $(w, w_{in})$ onto stack $S$
(12)             $w' \leftarrow$ value of first element of pertinentRoots of $w$
(13)             $(x, x_{in}) \leftarrow$ GetActiveSuccessorOnExternalFace($w'$, 1)
(14)             $(y, y_{in}) \leftarrow$ 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 $(w', w'_{out})$ onto stack $S$

(22)          else if $w$ is inactive,
(23)             $(w, w_{in}) \leftarrow$ GetSuccessorOnExternalFace($w$, $w_{in}$)

(24)          else assertion: $w$ is a stopping vertex
(25)               if the lowpoint of $c$ is less than $v$ and stack $S$ is empty,
(26)                  EmbedShortCircuitEdge($v'$, $v'_{out}$, $w$, $w_{in}$)
(27)               break the 'while' loop

(28)     if stack $S$ is non-empty, break the 'for' loop
```