Simplified $O(n)$ Algorithms for Planar Graph Embedding, Kuratowski Subgraph Isolation, and Related Problems

By

John M. Boyer
B.Sc., University of Southern Mississippi, 1990

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. Wendy Myrvold, Supervisor (Department of Computer Science)

Dr. John Ellis, Departmental Member (Department of Computer Science)

Dr. Frank Ruskey, Departmental Member (Department of Computer Science)

Dr. Gary MacGillivray, Outside Member (Department of Mathematics and Statistics)

Dr. S. Gill Williamson, External Examiner (Department of Computer Science and Engineering, University of California, San Diego)

Supervisor: Dr. Wendy Myrvold

ABSTRACT

A graph is *planar* if it can be drawn on the plane with vertices at unique locations and no edge intersections. 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 dissertation presents a new method for $O(n)$ planar graph embedding which avoids some of the complexities of previous approaches. The PC-tree method of Shih and Hsu has similarities to our algorithm, but the formulation is incorrect and not $O(n)$ for reasons discussed in this dissertation. Our planarity algorithm operates directly on an adjacency list representation of a collection of planar biconnected components, adding one edge at a time to the embedding until the entire graph is embedded or until a non-planarity condition arises. If the graph is not planar, a new $O(n)$ algorithm is presented that simplifies the extraction of a Kuratowski subgraph (a subgraph homeomorphic to $K_{3,3}$ or K_5). The results are then extended to outerplanar graphs, which are planar graphs that can be embedded with every vertex along the external face. In linear time, the algorithms find an outerplanar embedding or a minimal obstructing subgraph homeomorphic to $K_{2,3}$ or K_4. Finally, modifications to the outerplanarity and planarity obstruction isolators are presented, resulting in $O(n)$ methods for identifying a subgraph homeomorphic to $K_{2,3}$ or $K_{3,3}$.

Examiners:

Dr. Wendy Myrvold, Supervisor (Department of Computer Science)

Dr. John Ellis, Departmental Member (Department of Computer Science)

Dr. Frank Ruskey, Departmental Member (Department of Computer Science)

Dr. Gary MacGillivray, Outside Member (Department of Mathematics and Statistics)

Dr. S. Gill Williamson, External Examiner (Department of Computer Science and Engineering, University of California, San Diego)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I am particularly grateful to Dr. Wendy Myrvold, who provided outstanding intellectual guidance throughout this research. I have learned and grown immeasurably thanks to the many hours you devoted to our collaboration on this work.

I would like to thank my committee members, Drs. Wendy Myrvold, John Ellis, Gary MacGillivray, Frank Ruskey and S. Gill Williamson, for their valuable time spent considering this work and helping me to produce the best possible results. I have also appreciated and benefited from the great depth and breadth of computer science knowledge and style imparted by Drs. Wendy Myrvold, John Ellis, and Frank Ruskey, from whom I have had the privilege of enjoying many superb graduate courses. In this regard, I would also thank Dr. Valerie King for her scholarly teaching. I am fortunate to have called each of you my 'professor.'

I would like to thank my wife, Dr. Wanda A. R. Boyer, for her help and encouragement with matters mundane and sublime. To her I am most deeply grateful and indebted. In so many instances when I needed to talk to her about this work or about the process, or when I just needed time to complete this work, she supported me with innumerable acts of a kind and generous spirit. As well, I would like to thank my daughter, Wanda B. K. Boyer, for always wearing a smile, offering a pat on the back, and for learning the difference between a $K_{3,3}$ and a K_5. Thank you both for the lives we share together.

I would like to thank my parents, Daniel J. Boyer, Jr. and Kathleen Y. Boyer, who live very far away, for always being happy to hear from me, yet always being satisfied with whatever I could manage to do to keep in contact during this time.

Finally, I would like to thank my *other* Mom and Dad, Barbara Rumson and Gordon E. Rumson, for giving freely of themselves. Thank you for so many years of Friday night spaghetti and so many Saturdays free to think and create. Especially to Barbara ... Mom... thank you for helping me to 'tend my garden.'

Victoria, British Columbia, Canada — John M. Boyer

June 27, 2001

DEDICATION

To my wife and daughter, Wanda and Wanda.

1. Introduction

A *graph* $G = (V, E)$ consists of a set V of vertices and a set E of edges, each of which is associated with a pair of vertices from V. By convention, n indicates the number of vertices in V, and m indicates the number of edges in E. For a given graph G, the vertex set V, the edge set E and their sizes, n and m, can be denoted $V(G)$, $E(G)$, $n(G)$ and $m(G)$. A graph is *undirected* if its edges consist of unordered pairs of vertices, and a graph is *directed* if its edges consist of ordered pairs of vertices. Throughout this dissertation, graphs are undirected unless stated otherwise.

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 [24]), and a *simple graph* is a graph that forbids loops and multiple edges. Throughout this dissertation, graphs are 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 graph is often drawn using points for the vertices and lines (possibly curved) for the edges. A *planar representation* of a graph is a drawing of the 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 [24, p. 68]. A region of finite area is called a *proper face*. The *external face* is the plane less

the union of the proper faces and the points occupied by the planar representation of the graph. Figure 1.1(a) shows an example graph with four vertices and six edges. Figure 1.1(b) shows a planar representation of the same graph.

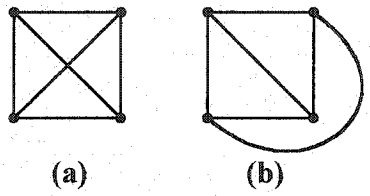


Figure 1.1: An Example Graph and a Planar Representation of the Graph

Graphs are often used as the underlying mathematical model in a wide variety of problems in which there are objects, represented by vertices, and associations between two objects, represented by edges. For example, a given electronic circuit may be composed of resistors, capacitors and transistors connected together by wires. A map of a website or a hypertext book could be represented using vertices for the pages and edges for the hyperlinks. In these example applications, finding a planar representation of the graph is useful as an answer to whether an electronic circuit can be arranged on a printed circuit board without creating a short-circuit [32], or whether a map or content overview of a website or hypertext book can be drawn on a computer screen without ambiguities arising from crossed edges.

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. Similarly, a graph is *outerplanar* if it is possible to create a planar representation of the graph in which all vertices are endpoints of the bounding edges of the external face. A *planarity testing algorithm* determines if a graph has a planar representation. A *planar embedding algorithm* also indicates the clockwise order of the neighbors of each vertex. 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.

In 1930, Kuratowski [41] was the first to prove a theorem characterizing the property of planarity in graphs, which had the effect of showing 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 *Kuratowski subgraph* is a minimal non-planar subgraph of a graph. A *Kuratowski subgraph isolator* is an algorithm that extracts a Kuratowski subgraph from a given non-planar graph. 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, the crossed edge in an identified Kuratowski subgraph could be replaced by a subcircuit of exclusive-or gates [43].

Existing linear time methods for the algorithms defined above are quite complex [11, 33]. This dissertation provides new, simplified linear time algorithms for planarity testing, planar embedding and Kuratowski subgraph isolation. The results are then extended to outerplanar graphs and to other related problems.

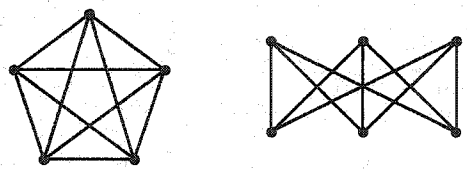
1.1 Further Definitions and Preliminaries

This section begins by defining the terms necessary to understand Kuratowski's famous theorem mentioned above. This is followed by basic graph theoretic terminology required to understand the overview of existing and related algorithms in Section 1.2 as well as the new algorithms presented in this dissertation.

A *walk of length* k is a sequence $P = v_0, e_0, v_1, e_1, v_2, \ldots, e_{k-1}, v_k$ of alternating vertices and edges such that $e_i = (v_{i-1}, v_i)$ is an edge of G for i from 1 to k. A *path* is a walk with no repeated vertex. A *cycle* is a walk of length greater than two with no repeated vertices except $v_0 = v_k$. In a simple graph, listing the edges is not necessary since the connecting edges are evident given the vertices.

A *complete graph* on n vertices, denoted K_n, is a simple undirected graph in which every pair of vertices is adjacent. A *complete bipartite graph*, denoted $K_{r,s}$,

is a simple undirected graph in which the vertex set V can be partitioned into two subsets V_r and V_s, of sizes $r > 0$ and $s > 0$ respectively, such that every edge has one endpoint in V_r and the other in V_s. For examples, the graph shown in Figure 1.1 is a K_4, and Figure 1.2 illustrates K_5 and $K_{3,3}$.



(a) K_5 (b) $K_{3,3}$

Figure 1.2: The Planar Obstructions K_5 and $K_{3,3}$

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* is a replacement of 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). If the resulting graph is to be kept simple, then the edge would only be added if it does not already exist. 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 was the first to characterize planar graphs by proving Theorem 1.1 below. Not only must planar graphs be devoid of subgraphs homeomorphic to K_5 and $K_{3,3}$, but non-planar graphs must contain a subgraph homeomorphic to K_5 or $K_{3,3}$. Thus, a Kuratowski subgraph isolator must return a 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.

Theorem 1.1 (Kuratowski [41]) *A graph is planar if and only if it contains no subgraph homeomorphic to $K_{3,3}$ or K_5.*

Seven years after Kuratowski's result, Wagner [58] proved a second characterization of planarity. 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)). In a combinatorial representation that maintains the cyclic order of the edges incident to each vertex, the cyclic order for the edges incident to w is equal to the combined cyclic order of the edges about the edge (u, v) before its contraction. A graph G contains a graph H as a *minor* if a graph isomorphic to H can be created from G by applying zero or more edge contractions to a subgraph of G. A graph H is a minor of G if G contains H as a minor. Wagner [58] characterized planarity using graph minors of $K_{3,3}$ or K_5, as stated in Theorem 1.2.

Theorem 1.2 (Wagner [58]) *A graph G is planar if and only neither $K_{3,3}$ nor K_5 are minors of G.*

Despite the early results of Kuratowski [41] and Wagner [58], the first characterization of planarity that lead to a polynomial time planarity testing algorithm is due to Tutte [55]. Given a subgraph H of a graph G, an *H-bridge* of G is either 1) an edge not in H whose endpoints are in H or 2) a component of $G - V(H)$ together with the edges (and vertices of attachment) that connect it to H. The *attachment points* of a bridge are those vertices of the bridge that are in $V(H)$. Some texts (e.g. [24]) require an H-bridge to have more than one attachment point. Given a cycle H in a graph G, two H-bridges *conflict* if they have three common attachment points or if there are four distinct vertices v_1, v_2, v_3 and v_4 in cyclic order on H such that v_1 and v_3 are attachment points in the first bridge and v_2 and v_4 are attachment points in the second bridge. The *conflict graph* of H is a graph in which each vertex represents an H-bridge of G and each edge (u, v) represents a conflict between the H-bridges represented by u and v. According to Tutte, a graph G is planar if and only if, for every cycle C in G, the conflict graph of C is bipartite. This result was exploited in the first polynomial time planarity testing algorithm created by Auslander and Parter

[5] (and correctly formulated by Goldstein [25]).

A graph is *connected* if, for every pair of vertices u and v, there exists a path from u to v in the graph. A graph is *disconnected* if there exist vertices u and v for which there is no path from u to v in the graph. A *connected component* is a maximal connected subgraph of a graph. A vertex v is a *cut vertex* of a graph G if the removal of v and its incident edges results in a graph with more connected components than G. Another name for a cut vertex is an *articulation point.* A *cut edge* is an edge whose removal increases the number of connected components in G. Each endpoint of a cut edge is either a cut vertex or a vertex of degree one. A graph containing one or more cut vertices is *separable*, and a graph with no cut vertices is *biconnected.* A *biconnected component* of a graph is a maximal biconnected subgraph. Another name for biconnected component is *block.* Figure 1.3(a) illustrates a cut vertex v separating two biconnected components, and Figure 1.3(b) shows the two components that result when v and its incident edges are removed. Note that v would no longer be a cut vertex if edge (u, w) were added.

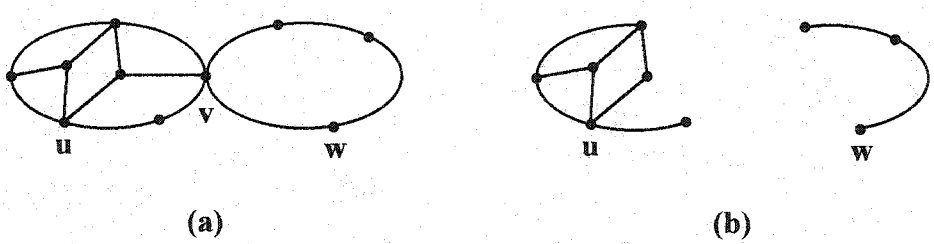


Figure 1.3: A Cut Vertex v Separating Two Biconnected Components

Since most planarity algorithms exploit some property of biconnected graphs, the input graph is assumed to be biconnected. When the input graph is not biconnected, a preprocessing step identifies the biconnected components of the input graph, passing each as a subgraph to the planarity algorithm. This is an acceptable practice since a Kuratowski subgraph, if it exists, must be a subgraph of a single biconnected component. Planar embedding algorithms require an additional postprocessing step to assemble the embeddings of the biconnected components into a single embedding of the input graph, but this is a simple task since the biconnected components that

must be attached to a cut vertex can be arbitrarily permuted.

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 [53]. Depth first search (DFS) on graphs operates in the same way as the well-known pre-order traversal of a tree, 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, 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.

A post-order traversal of the depth first search tree can be used to compute the well-known function *lowpoint(v)* (defined in [53]), which associates with each vertex v the least DFI of any vertex reachable from v by a sequence of zero or more tree edges to descendants of v followed by at most one back edge. The lowpoint function can then be used to characterize cut vertices, as shown by Theorem 1.3.

Theorem 1.3 (Tarjan [53]) *A vertex v is a cut vertex if and only if v has a DFS child c with $lowpoint(c) \geq DFI(v)$ and there exists a distinct vertex w not a descendant of c in the same connected component as v.*

Proof. Because the endpoints of a back edge join a descendant with an ancestor in the DFS tree, the constraint $lowpoint(c) \geq DFI(v)$ implies that every back edge

incident to a vertex in the DFS subtree rooted at c has for its other endpoint either v or another vertex in the DFS subtree rooted at c. Thus, every path from c to w must contain v. Since removing v destroys all paths between c and w, v is a cut vertex. Conversely, suppose v is a cut vertex. If v is the DFS tree root, then every non-descendant of c other than v must be in other DFS subtrees rooted by other DFS children of v. Any DFS child of v other than c is a suitable choice for w. On the other hand, if v is not the DFS tree root, then any ancestor of v is a suitable choice for w. □

Given the cut vertices of a graph, determining the biconnected components is a straightforward process that again involves the depth first search tree. Consider a tree edge (v, c) in which v is determined to be a cut vertex according to Theorem 1.3. Based on the proof of this theorem and the definition of biconnected component, the biconnected component B containing (v, c) has the following properties: 1) v is the vertex with the least DFI in B, 2) the vertex subset $V(B)$ contains v and every vertex d in the subtree rooted by c that has no ancestor cut vertex separating d from c along the tree path from d to c, and 3) the edge subset $E(B)$ contains all edges of the graph with both endpoints in $V(B)$. These properties are easily exploited during a post-order traversal of the depth first search tree to produce the biconnected components of the graph. In fact, Tarjan's original algorithm [53, p. 153] calculates the DFI and lowpoint of each vertex and determines the edges in each biconnected component using a single depth first search.

A biconnected component that is planar and contains more than one edge has an external face whose bounding edges form a cycle, just as is the case for proper faces. The bounding edges of a face, together with their vertex endpoints, form the *bounding cycle* of the face. Because combinatorial planar embeddings consist of vertices and edges rather than regions, the term 'external face' typically refers to the bounding cycle of the external face of a biconnected component larger than a single edge, or to the walk $(v, (v, w), w, (w, v), v)$ for a biconnected component consisting only of the edge (v, w).

Theorem 1.4 states Euler's formula, which relates the number of faces f in a planar representation of a connected planar graph to the number of edges m and vertices n in the graph. Euler's formula is easily extended to $f = m - n + 1 + c$ for disconnected simple planar graphs with c connected components, which can be seen by adding cut edges to join the DFS tree roots. The result is a well-known corollary that bounds the number of edges in any planar graph to be a constant multiple of the number of vertices, as stated in Corollary 1.5.

Theorem 1.4 (Euler [17]) *If G is a connected simple planar graph with n vertices and m edges, then every planar representation of G has a number of faces f equal to $m - n + 2$.*

Corollary 1.5 *If G is a simple planar graph with $n > 2$ vertices, then G contains no more than $3n - 6$ edges.*

Due to Corollary 1.5, algorithms for planarity testing, planar embedding and Kuratowski subgraph isolation on G (with $n > 2$) can be restricted to any subgraph H containing $\min(3n - 5, |E(G)|)$ edges from G. If G exceeds the edge limit, then H is guaranteed to be non-planar by Corollary 1.5, so it contains a Kuratowski subgraph by Theorem 1.1. Thus, a planarity tester, planar embedder or Kuratowski subgraph isolator that performs constant work per edge achieves $O(n)$ time.

1.2 Review of the Literature

The examples at the beginning of this chapter, which concerned the planarity of electronic circuits and website maps, were representative of a rather straightforward application of planarity algorithms. There exist numerous applications of a more subtle variety. For example, a number of algorithms have been designed specifically to exploit properties of planar graphs, so a planarity tester or embedder should precede their operation since they do not work unless the graph is planar. Examples include linear-time planar graph five-coloring algorithms [12, 22, 44], polynomial time planar graph four-coloring algorithms [1, 2, 3, 50, 54], and the delta-wye reduction algorithm of Feo and Provan [21].

There also exist problems that are very difficult on graphs in general but which are solved much more easily on planar or outerplanar graphs, such as graph isomorphism [27, 29, 31]. For example, a planar graph can be used to model the significant molecular structure of most known chemical compounds using vertices to represent atoms (or groups of atoms) and edges to represent chemical bonds. Thus, determining the equivalence of two chemicals can often be reduced to the planar graph isomorphism problem [27]. Another example is finding the maximum cut in a graph, which is NP-complete in general [23] but solved efficiently on planar graphs [26]. In the case of outerplanar graphs, an example is the algorithm of Ellis, Mata and MacGillivray [16] for finding the longest path between two vertices of an outerplanar graph, which achieves linear time on a problem that is NP-Complete even for planar graphs [23].

As well, efficient Kuratowski subgraph isolation is a critical component of numerous linear-time graph algorithms, including an isolator for $K_{3,3}$ homeomorphs [20], an isolator for K_5 minors [37], and embedding algorithms for more complex surfaces than the plane [34, 46, 47]. For example, the linear time $K_{3,3}$ homeomorph isolator of Fellows and Kaschube [20] begins by isolating a Kuratowski subgraph K, which is either the desired $K_{3,3}$ homeomorph or a K_5 homeomorph. In the latter case, the algorithm examines the K-bridges of the graph, both recursively and to determine whether any can be attached to K to form a $K_{3,3}$ homeomorph.

The wealth of interest in graph planarity has resulted in efficient algorithms for various computing models and constraints, including both parallel [38] and on-line [6] planar embedders. In addition, various efficient algorithms have been developed for enumerating, ranking and unranking planar embeddings [10, 36, 57].

The first planarity testing algorithm to achieve linear time began with the work of Auslander and Parter [5] (correctly formulated by Goldstein [25]). The algorithm obtained a cycle C in the graph, recursively embedded each C-bridge (augmented with edges to form a path from its first to last attachment point along C), then determined an embedding in which some of the bridges were attached inside C while the conflicting bridges were attached outside of C. Hopcroft and Tarjan [30] optimized the algorithm to linear time by exploiting numerous properties of depth first search

that allowed them to decompose each bridge into a set of paths to be processed consecutively by the algorithm. As a result, a number of authors refer to the Hopcroft and Tarjan algorithm as the *path addition method* [11, 18, 52]. Some corrections to the algorithm appear in [15], and significant additional details are presented by Williamson [61, 63] as well as the text by Reingold, Nievergelt and Deo [49].

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 bridges (sets of paths) on the inside or outside of 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 the conclusion of their paper briefly sketches a method for augmenting their 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" [11, p. 55].

Another planarity test created by Demoucron, Malgrange and Pertuiset [14] (see [24] for an excellent exposition) also begins with a cycle C and the bridges with respect to C. Then, it embeds each bridge one at a time, keeping track of the new faces that develop in the embedding and the faces into which each remaining bridge can be placed. Rubin [51] describes a method for optimizing this method to achieve average case linear time and roughly half the running speed of the Hopcroft and Tarjan method. However, these results are based on empirical tests and no proof is given that the algorithm achieves linear time in the worst case.

The second method of planarity testing proven to achieve linear time began with an algorithm some have called the *vertex addition method.* The original $O(n^2)$ version is due to Lempel, Even and Cederbaum [42]. The algorithm begins by creating an s,t-numbering for the (biconnected) graph. A property of this 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 of the first k vertices such that the remaining vertices ($k+1$ to t) can be embedded in a single face. The testing

algorithm was optimized to linear time by a pair of contributions. Even and Tarjan [19] optimized s,t-numbering to linear time, while Booth and Lueker [7] 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 [11] augmented the PQ-tree operations so that a planar embedding is computed as the vertex addition method operates.

Achieving linear time with the vertex addition method is also quite complex [33], in part because Booth and Lueker do not include the complete set of optimized templates required to update the PQ-tree quickly [7, 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 in a reduction.

Linear time Kuratowski subgraph isolation was first solved by Williamson [62]. The method examines the state of the data structures in the Hopcroft and Tarjan path addition method at the point when a non-planarity condition occurs (when three bridges mutually conflict). This algorithm is also quite complex, in part because it is based on path addition. Karabeg [35] developed a linear time Kuratowski subgraph isolator that exploits non-planarity conditions that arise in PQ-trees. A far simpler Kuratowski subgraph isolator was developed by Klotz [39]. It only achieves $O(n^2)$ time, but it is useful because it does not dependent on a complex linear time planarity test algorithm.

A characterization of planar graphs by de Fraysseix and Rosenstiehl [13] could lead to algorithms for planarity testing and Kuratowski subgraph isolation, though the paper contains no development of a linear time methodology. The characterization is important, though, because it discusses planarity from the vantage point 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 [8]. Our work is based on this bottom-up view, though we derived it independently

of [13] 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 [8] 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 dissertation 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.

Very recently, a new planarity test was presented by Shih and Hsu [52]. They develop the notion of a PC-Tree (described below) as a simplification of a PQ-tree, and their work is critically dependent on a number of the same results that appeared earlier in Boyer and Myrvold [8]. However, since their date of submission was significantly before the publication of [8], there is clearly a case of simultaneous independent discovery of somewhat similar algorithms.

Unfortunately, the results of Shih and Hsu stated in [52] contain several errors that negate the proof of correctness of the planarity test and the claim of a fully defined Kuratowski subgraph isolator. For example, the only PC-tree pattern in [52] for detecting K_5 homeomorphs appears to only detect some K_5 minors. According to [52, 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". Figure 1.4(a) depicts the K_5 minor pattern appearing in Figure 6(c) of [8] (with a few cosmetic changes), and Figure 1.4(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

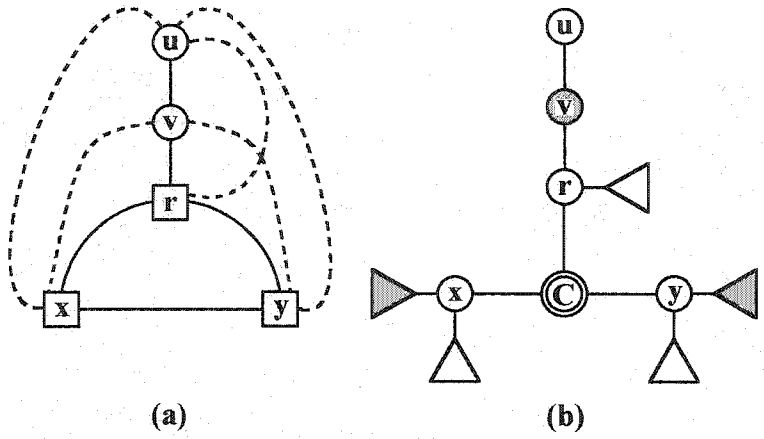


Figure 1.4: The K_5 non-planarity minor from [8], and the corresponding PC-tree at the beginning of step v

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 [52, p. 181]). However, 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 [52] 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 [52], 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 [52] 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 [52] clearly indicate how to continue processing under the configuration given in Figure 10 of [52], 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 [8], which is depicted (with a few cosmetic

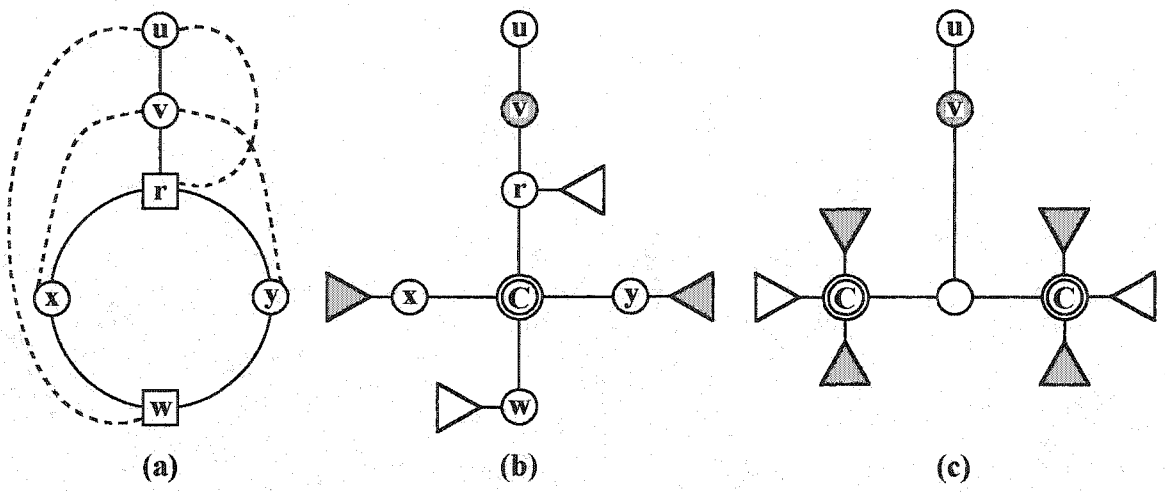


Figure 1.5: (a) One of the $K_{3,3}$ non-planarity minors from [8], (b) A corresponding PC-tree at the beginning of step v, (c) Another example of the PC-tree problem with two terminal nodes

changes) in Figure 1.5(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 1.5(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 [52] contains no theorem, lemma or corollary that detects a non-planarity condition in this configuration.

In essence, the non-planarity pattern depicted in Figure 1.5 is a counterexample to the proof of correctness appearing in Theorem 4.1 of [52]. Any reduction of the PC-tree in Figure 1.5(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 [52] that the conditions established by "Lemma 2.5, Corollary 2.6 and Lemmas 3.1 and 3.2 ... imply a feasible internal embedding 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 [60], this problem was determined to also be applicable when there are two terminal nodes, as depicted in Figure 1.5(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 [52]. 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 [52] also have an effect on the complexity analysis. In accounting for Lemma 3.1, p. 190 of [52] 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 [52]. For example, the claim that the "RBC will be stored as a circular doubly linked list" [52, 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 [8].

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 [52, p. 184]: "we shall represent the 2-connected component by a C-node whose parent is i ...". 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 dissertation and of [8] in order to create an algorithm that achieves linear time while exploiting those graph-theoretic properties that are common to both algorithms.

1.3 Contributions of This Dissertation

This dissertation contributes new planarity testing and embedding algorithms, with rigorous proofs of correctness and linear time performance, that operate directly on data structures used to represent combinatorial planar embeddings. Sufficient information is included on how to maintain linear time performance while following the correct paths through biconnected components, flipping biconnected components as necessary, and producing a final, consistent orientation for all vertices. The length of discourse is reflective both of the decision to pursue a depth of rigor in proof, analysis and software specification not typically found in a textbook treatment of linear time planarity (such as in [49]) and of the numerous additional related problems addressed in this dissertation.

One such problem is that of isolating a Kuratowski subgraph in a non-planar graph. Williamson [62] states that "it would be desirable to have not one but several basically different [linear time Kuratowski subgraph isolators]" because the condition of linearity "forces the emergence of a certain level of insight into the structure of non-

planar graphs and Kuratowski's theorem." This dissertation contributes a new linear time Kuratowski subgraph isolator that differs from and is more intuitive than prior strategies described by Boyer and Myrvold [8], Shih and Hsu [52], and Williamson [62]. The algorithm is conceptually as simple as the Kuratowski subgraph isolator of Karabeg [35], though our implementation details are not as complicated, in part because they do not involve the augmentation of PQ-tree operations.

A linear time implementation of these algorithms was created in four days for the planar embedding algorithm plus 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 [45] (specifically all connected graphs with 11 or fewer vertices). For each graph, the integrity of the resulting combinatorial planar embedding or Kuratowski subgraph was tested.

In this dissertation, the new theoretical framework developed for planarity is also extended to produce a linear time outerplanarity testing algorithm distinct from the algorithm of Brehaut [9], which is based on the Hopcroft and Tarjan planarity test [30]. Moreover, an outerplanar embedding is recovered, or a minimal obstructing subgraph homeomorphic to $K_{2,3}$ or K_4 is isolated if a graph is found not to be outerplanar. Finally, modifications to the outerplanar and planar obstruction isolators are described, resulting in linear time algorithms that search a graph to identify a $K_{2,3}$ or $K_{3,3}$ homeomorph or determine that the graph has no subgraph homeomorphic to $K_{2,3}$ or $K_{3,3}$.

1.4 Overview of Chapters

The previous discussion described how the results in this dissertation relate to other work. We now give an overview of the dissertation, which summarizes the original results demonstrated in the remaining chapters.

Chapter 2 presents a top-down overview of the new planarity algorithms, and Chapter 3 presents the data structures used by the new planarity algorithms. Chapters 4 through 7 present the planarity test algorithm details in a bottom-up fashion,

including proofs of correctness and run-time analyses for each functional component. Chapter 7 presents the key theorems proving the correctness and linear time operation of the new planarity testing algorithm. Chapter 8 describes the data structure and algorithm modifications necessary to extract a combinatorial planar embedding if the input graph is planar, and it proves that these modifications result in a linear time planar embedding algorithm. Chapter 9 describes how to extract a Kuratowski subgraph in linear time from the state of the data structures at the moment when the planarity tester discovers that the input graph is non-planar. Chapter 10 extends the theoretical framework developed in this dissertation for planarity problems to solve the corresponding outerplanarity problems, and Chapter 11 describes modifications to the outerplanarity and planarity obstruction isolators that result in linear time algorithms that search a graph for a $K_{2,3}$ or $K_{3,3}$ homeomorph. Finally, Chapter 12 discusses several future lines of inquiry that can be pursued with the new data structures and algorithms in this dissertation.

2. Overview of New Planarity Algorithms

Our new planarity algorithm begins with a few preprocessing operations, most of which are common to all linear time planarity algorithms. The first operations are to create a depth first search tree and to perform the lowpoint calculation as described in Chapter 1. This assigns a depth first index (DFI) and lowpoint to each vertex. The vertices are then sorted into ascending DFI order (in linear time). A few additional data structures and preprocessing operations are required, which will be described as needed in this chapter, then in more detail in Chapter 3.

After preprocessing the input graph G, the algorithm creates a data structure $\tilde{G}$ to store the embedding. The data structure $\tilde{G}$ is designed to maintain a collection of biconnected components. For each DFS tree edge e of G, the algorithm creates a singleton biconnected component in $\tilde{G}$ containing only a representation of the tree edge e, including a structure for each vertex incident to e. Thus, for each vertex v, $\tilde{G}$ contains a vertex structure for v for each tree edge containing v. In the main loop of the algorithm described below, the multiple vertex structures representing a vertex v are merged together as back edges are embedded to form larger biconnected components.

The main loop of the algorithm iterates through each vertex v of G in reverse DFI order, embedding in $\tilde{G}$ one edge per back edge between v and a descendant of v in G. The rationale for using 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 they each have a path of DFS tree edges leading to the DFS tree root. A high-level outline for our algorithms appears in Figure 2.1.

As indicated in the main loop body in Figure 2.1, embedding a back edge (v, w) may be accompanied by merging one or more biconnected components that, together with (v, w), form a single biconnected component. The order in which back edges are embedded and the details of the biconnected component merge operations are selected

Figure 2.1: High-Level Outline of Planarity Algorithms

(1) Receive graph G with $n > 2$ vertices and $m \leq 3n - 5$ edges
(2) Perform depth first search and lowpoint calculations on G
(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 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

such that all vertices with unembedded back edges to v or its ancestors remain on the external faces of biconnected components in $\tilde{G}$.

If the input graph G is planar, then all of the back edges are embedded in the main loop. Since all of the tree edges were embedded in Line 4, $\tilde{G}$ contains a planar embedding of G. However, if G is non-planar, then the algorithm terminates in some step v when it cannot embed a back edge to v without also placing a vertex with an unembedded back edge to a DFS ancestor of v inside of a biconnected component (such that the unembedded back edge would have to cross the bounding cycle of the biconnected component in order to be embedded). When this occurs, our algorithm adds a few unembedded back edges, then isolates a Kuratowski subgraph in $\tilde{G}$.

The sections of this chapter describe a more detailed conceptual framework for

the algorithm outline given in Figure 2.1. The first few sections discuss preliminary details of the data structures, while the remaining sections describe the algorithm operation in more detail, adding to the data structures as necessary. The formal data structure definition appears in Chapter 3 and the formal descriptions for the procedures in the algorithm appear in subsequent chapters (along with proofs of correctness and run-time analyses).

2.1 Modified Adjacency List Representation

The data structure used to represent the embedding of the input graph is quite similar to the standard adjacency list format normally used to represent a combinatorial planar embedding. There is a vertex structure to represent each occurrence of a vertex in a biconnected component. The adjacency list of a vertex is represented by a list of edge records, with one edge record for each edge incident to the vertex. An edge (u, v) is represented by a pair of edge records e_u and e_v. The edge record e_u in u's adjacency list indicates v as a neighbor of u, and the edge record e_v in v's adjacency list indicates u as a neighbor of v. To help with traversing an edge in constant time, the edge records e_u and e_v indicate one another using a *twin link*. To conserve memory, the twin link between two edge records can be made implicitly by storing the edge records in consecutive memory locations.

The embedding data structure, denoted $\tilde{G}$, has an array V of vertex structures and an array E of edge records. The vertices from G are represented in $\tilde{G}$ by vertex structures in the array V in the order given by the DFI of the vertices (more details regarding storage of extra copies of cut vertices appears in Section 2.2). The vertex structures contain other important information copied from G such as the lowpoint and DFS parent of the vertex, but the adjacency list of each vertex structure is initially empty. Edge records are created in array E of $\tilde{G}$ as edges are embedded in Lines 4 to 6 of Figure 2.1. The vertex structures and edge records maintain additional information as the embedding algorithm proceeds. These additions are described as needed throughout the rest of this chapter. Based on the discussions in this chapter, the formal data structure definition appears in Chapter 3.

2.2 Maintaining a Collection of Biconnected Components

The *root vertex* of a biconnected component is the vertex with the least DFI in the biconnected component. A *child biconnected component* of a vertex v is a biconnected component in which v has the least DFI. The *parent biconnected component* of a cut vertex v is the biconnected component that contains v as well as the DFS parent of v. A DFS tree root has at least one child biconnected component for which it is the root vertex, but it has no parent biconnected component since it has no DFS parent. All vertices in a DFS tree other than the root have a parent biconnected component, and each cut vertex has one or more child biconnected components.

In a biconnected component B with a root vertex v, there exists only one DFS child c of v in B. The assumption of a second 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 the root vertex v. Since v only has one DFS child c in B, the tree edge (v, c) is defined to be the *root edge* of B.

In our data structures, cut vertices are represented by using one vertex structure for each occurrence of a vertex in some biconnected component. For each child biconnected component B, the root vertex v of B is represented by a *root copy* of v. The *parent copy* of v appears in B's parent biconnected component. The parent copy of a vertex is the principal representative of the vertex (except for each DFS tree root, whose root copies are merged to form a parent copy only after the embedding is finished). Each root copy contains a *parent link* to indicate its parent copy vertex. A root copy of v can be denoted v^c, where c is the DFS child of v in B. In most cases, only one root copy is under consideration, and it is not necessary to specify the DFS child. In these cases, a root copy of a vertex v is denoted v'.

The vertex structure array V in $\tilde{G}$ contains n locations for parent copies and another n locations for root copies. Since a root copy v^c can be uniquely associated with a DFS child c of v, the array location $V[n + c]$ can be used to represent v^c without conflicts.

Before any back edges are embedded, the algorithm adds each DFS tree edge (v, c) as a singleton biconnected component containing only (v^c, c). An example of

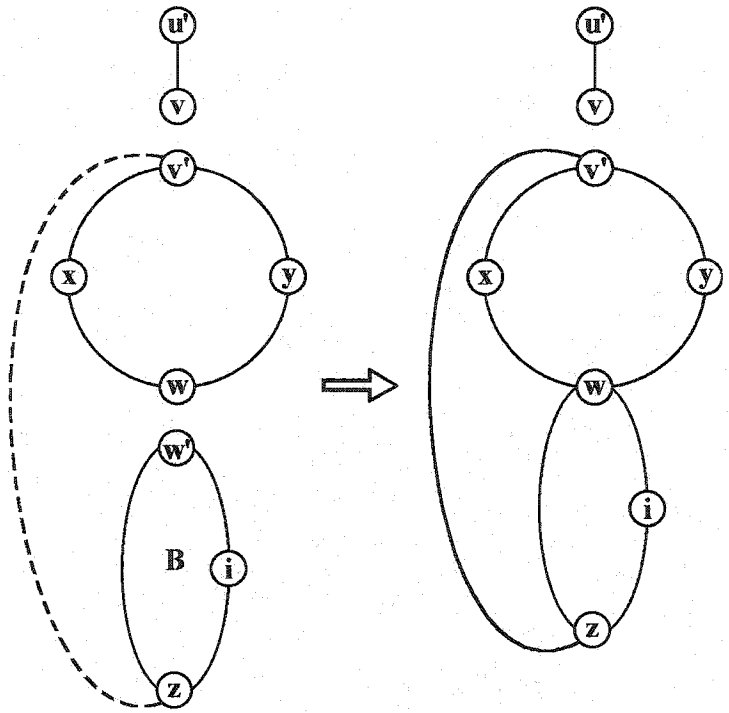


Figure 2.2: High-Level View of Merging a Biconnected Component

this is the edge (u', v) in Figure 2.2. Thus, the first edge embedded for a biconnected component is its root edge.

When the algorithm adds a back edge, if the endpoints are in separate biconnected components, then the necessary biconnected components are merged together to form a single connected component, which becomes biconnected when the back edge is added. In Figure 2.2, this first occurred in step w when the back edge (w, z) was added to form biconnected component B. In the current step v depicted by Figure 2.2, the algorithm is ready to embed the back edge (v, z). Since vertex w will no longer be a point of separation between a child biconnected component B and its parent biconnected component once the edge is added, the root vertex w' of B is merged with its parent copy w. Then, the representative edge for (v, z) is added, resulting in a biconnected component that includes z, w and a root copy of v.

The back edge (v, z) is embedded between z and a root copy of v, denoted v' in Figure 2.2. When processing v, the ancestors of v have not yet been processed, so there are no back edges leading to ancestor vertices of v. Hence, v is a cut vertex in

the current embedding, and all back edges from v to its descendants are embedded incident to root copies of v. Earlier, in step w, the back edge (w, z) was embedded by adding the edge (w', z) to form biconnected component labeled B in Figure 2.2. The edges incident to w' are only changed to be incident to w when we merge w and w' as part of embedding the back edge (v, z).

2.3 The External Face

In each biconnected component in $\tilde{G}$, our data structures distinguish a particular face as being the external face. To accomplish this, our data structures represent the adjacency list of a vertex structure for v by placing the vertex structure in a circular doubly linked list with the edge records indicating the neighbors of the vertex in the biconnected component. The edge records are linked according to their cyclic order about v, and the vertex structure for v is placed between the two edge records that appear on the external face of the biconnected component. To represent these links, each vertex structure and edge record contains two members, denoted link[0] and link[1], that indicate the predecessor and successor in the adjacency list. For a biconnected component containing a single tree edge (p, c), both links of the vertex structure p indicate edge record e_p, both links of e_p indicate p, e_p indicates c as the neighbor and edge record e_c by twin link. The analogous links are made for c and e_c.

According to this design, the algorithm begins a traversal from a vertex x to a vertex y along the external face of a biconnected component by way of the link from the vertex structure x to the edge record for (x, y). Next, the algorithm uses the twin link to reach (y, x). Finally, the algorithm uses the link in (y, x) that leads to the vertex structure for y. To proceed to the next vertex after y, the algorithm simply selects the link that does not indicate the edge record (y, x) leading back to x (except if y is degree one since both links indicate the edge record (y, x)).

2.4 External Activity

Given that our data structures maintain a collection of biconnected components as well as the external face of each biconnected component, progress with the high-

level algorithm outline in Figure 2.1 requires that we be able to determine which vertices and biconnected components must be kept on the external face as the back edges of a given vertex v are embedded.

Definition 2.1 *A parent copy vertex x is* externally active *if the input graph G contains a back edge (u, x) where u is an ancestor of the current vertex v being processed by the main algorithm loop, or if there exists a biconnected component in the embedding $\tilde{G}$ rooted by x' (a root copy of x) that contains at least one externally active parent copy vertex.*

Definition 2.2 *An* externally active biconnected component *is a biconnected component that contains at least one externally active parent copy vertex.*

Although biconnected components, which are identified by root copy vertices, can be externally active, there is no notion of external activity for root copies of vertices per se. A root copy vertex does not share the external activity status of its parent copy nor of the biconnected component for which it is the root, and the algorithms in this dissertation do not check root copy vertices for external activity.

We refer to a vertex x as 'externally active' to indicate that x plays an active role in the embedding of back edges to vertices with a lower DFI than the current vertex v, and therefore x must be kept on the external face as the back edges to the current vertex v are embedded.

If a vertex x is directly adjacent to an ancestor of the current vertex v, then certainly x is externally active. Figure 2.3(a) illustrates what happens if a back edge were added between the current vertex v and one of its descendants w such that the bounding cycle of the biconnected component containing v, w and x were to surround x. When the main loop iterates to vertex u, the back edge from x to u would be required to cross this bounding cycle, which contradicts the purpose of a planarity algorithm. Instead, our algorithms determine that x must remain on the external face, so the edge (v, w) is embedded such that y is surrounded, as shown in Figure 2.3(b). Thus, x can be still be found along the external face when the main

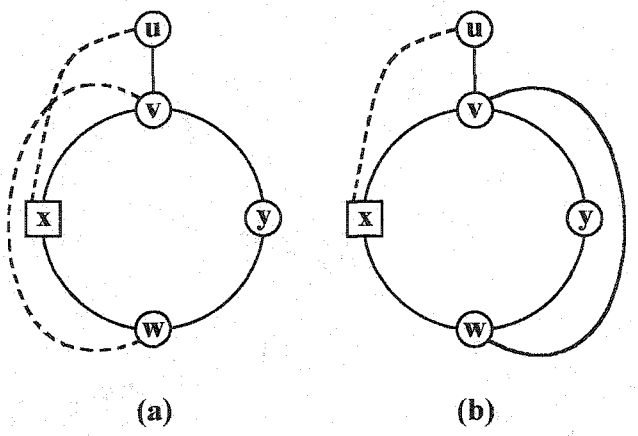


Figure 2.3: Embedding (v, w) While Keeping x on the External Face

loop processes u, and the back edge (u, x) can be embedded without crossing any edges.

As shown in Figure 2.4, a vertex x is also considered to be externally active if there exists a back edge (u, z) where u is a DFS ancestor of the vertex v currently being processed and z is a descendant of x in *another biconnected component* of $\tilde{G}$. Note that the biconnected components rooted by x' and d' are externally active due to vertices d and z, respectively. If the back edge (v, w) were embedded such that x appeared inside the cycle $C = [v, \ldots, w, \ldots, y, \ldots, v]$, then either the back edge (u, z) or a tree edge in the DFS tree path from z up to x would be required to cross C (which is why our algorithms do not place externally active vertices inside of the external face bounding cycles of biconnected components).

Because it is often unnecessary to distinguish why a vertex is externally active, many diagrams in this dissertation depict an externally active vertex x as being directly adjacent to an unprocessed vertex u unless the context demands that a descendant of x also be shown. Likewise, the edge (v, w) may be indicative of a path connecting v and w that includes descendants of w. In other words, all descendants are contracted into x or w when there is no conceptual difference between connecting u and x or v and w by a back edge or by a path of tree edges followed by a back edge. Moreover, the DFS ancestors of the current vertex v (both root copies and parent

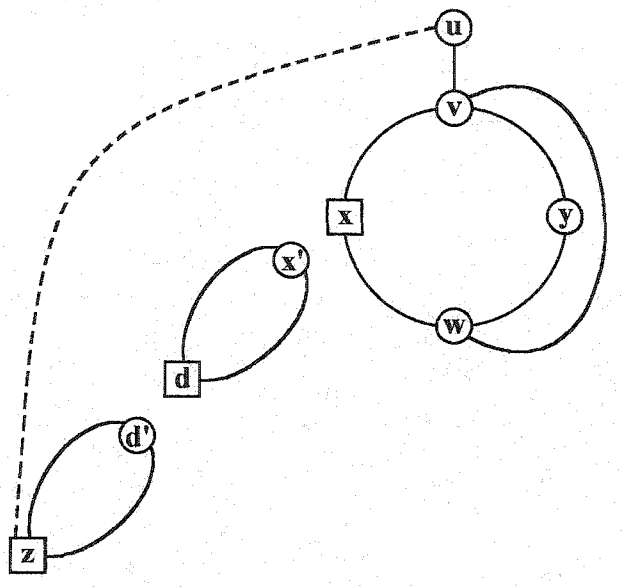


Figure 2.4: Due to z, Vertex x is Externally Active

copies) are contracted into vertex u, and edges such as (x, w) may represent paths along the external face whose edges and vertices have been contracted into x or w for discussion purposes. In essence, the diagrams are graph minors of the embedding that contain the vertices and edges necessary to illustrate a particular point.

Computationally, it is easy to determine in constant time whether a biconnected component is externally active using the lowpoint function described in Chapter 1. Given a biconnected component B rooted by the DFS child tree edge (r', c), B is externally active if the lowpoint of c is less than the DFI of the current vertex v whose back edges are being embedded.

Detecting whether a vertex is externally active is nearly as simple as using the lowpoint function, and lowpoint figures prominently in the computation. The lowpoint of w is the DFI of the vertex closest to the DFS tree root that is reachable by a back edge from w or *any* of its DFS descendants. According to Definition 2.1, the external activity computation for w excludes any DFS subtree rooted by a child c that has been merged into the parent biconnected component with w.

External activity can be computed in constant time using the following methods. During initialization of our data structures, each (parent copy) vertex structure receives a member *leastAncestor* to store the DFI of the least ancestor adjacent to

the vertex by a back edge, which is a value obtained during the lowpoint calculation. Each vertex structure also receives a doubly linked list of its DFS children sorted by their lowpoint values, which we call the *SeparatedDFSChildList* because a DFS child c is removed from the list of its DFS parent w when a biconnected component rooted by the DFS tree edge (w', c) is merged with w. Finally, each vertex structure receives a reference to indicate its representative node in the SeparatedDFSChildList of the DFS parent so that the removal can be done in constant time. A vertex w is externally active if its leastAncestor member is less than v or if the least lowpoint from the elements of its SeparatedDFSChildList, which is in the first element, is less than v, where v is the vertex currently being processed.

2.5 Walking Up to Prepare for Embedding of Back Edges

As described in the algorithm outline in Figure 2.1, the embedding of a back edge (v, w) may involve merging a number of biconnected components that, together with the new back edge, form a single biconnected component in the embedding. For each back edge in the input graph G that is incident to the current vertex v and a descendant w of v, our algorithm invokes a procedure called 'Walkup' to identify the biconnected components that must be merged before (v, w) can be embedded (see Line 6(a) of Figure 2.1). The Walkup also sets a flag in w to indicate that w is the descendant endpoint of a back edge that must be embedded during step v of the main algorithm loop.

Definition 2.3 *A biconnected component with root edge (r', c) is* pertinent *at step v if the input graph G contains at least one back edge (v, w), where w is in the DFS subtree rooted by c.*

Definition 2.4 *A vertex w is* pertinent *in step v if there exists a back edge (v, w) in the input graph G that has not been embedded in $\tilde{G}$ or if $\tilde{G}$ contains a pertinent biconnected component with root w'.*

Definition 2.5 *A vertex r is a* separation ancestor *of a vertex w if r is a DFS ancestor of w and a DFS descendant of the current vertex v, and if the parent copy of r is not in the same biconnected component as the parent copy of w in $\tilde{G}$.*

Based on Definitions 2.3, 2.4 and 2.5, the invocations of the Walkup procedure identify pertinent vertices and pertinent biconnected components at the beginning of step v. The members *pertinentBicompList* and *adjacentTo* are added to each vertex structure to store this information. At the beginning of step v, the adjacentTo member of each vertex structure is greater than v, and the pertinentBicompList of every vertex is empty. For a back edge (v, w), the Walkup sets the adjacentTo member of w equal to v. To identify the pertinent biconnected components, the Walkup procedure for back edge (v, w) performs a loop that begins at the parent copy of w and simultaneously traverses both paths leading from w around the external face of the biconnected component B containing w until the root vertex r' of B is encountered. If r' is a root copy of v, then the Walkup terminates successfully. Otherwise, r' is stored in the pertinentBicompList of the parent copy vertex structure r. Then, the Walkup loop reiterates starting at r. Thus, for each separation ancestor r of w in $\tilde{G}$, the root copy r^c of a biconnected component with c an ancestor of or equal to w is recorded in the pertinentBicompList of r. For the sake of simplicity, the root copy is appended if the corresponding biconnected component is externally active and prepended otherwise.

Within each biconnected component visited by Walkup, one of the two 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 between r' and w around the external face. This helps to ensure that the total cost of all Walkup operations is a constant multiple of the sum of degrees of proper faces in the embedding.

The Walkup loop has a second stopping condition due to a necessary optimization. In order to achieve linear time, the cumulative work done by all Walkup calls for the back edges of v must not exceed a constant multiple of the number of bounding edges in new proper faces formed during step v. However, it is possible for the back

edges of v to be in a processing order that would cause successive Walkup calls to traverse the same paths many times. Fortunately, a Walkup initiated for a given back edge to v can halt if it encounters any vertex structure z visited by a prior Walkup for a different back edge to v since the prior Walkup call will have already recorded the remaining pertinent biconnected components for separation ancestors between z and v. This optimization also prevents a child biconnected component from being recorded as pertinent more than once.

Each vertex structure w in our embedding structure also contains a member called *visited*, which is equal to v if and only if w has been visited by Walkup in step v. As Walkup visits each vertex, it assigns v to the 'visited' member of the vertex. Any future Walkup operations during step v can stop if a vertex with the visited member set to v is encountered. Moreover, these visitation flags are cleared at the end of step v with no extra work because the step variable of the main algorithm loop is decremented to begin processing the vertex with the next lower DFI.

Although Chapter 5 provides the details of the Walkup procedure, Figure 2.5 exemplifies the processing performed by Walkup. In this example, there exist back edges from v to vertices w and y. The vertices w' and z' are roots of non-pertinent biconnected components. The arrows indicate paths taken by the two Walkup calls, and the octagons indicate where each Walkup halts.

The Walkup for the first back edge, (v, w), starts at the parent copy of w and proceeds in parallel around the external face along both paths leading from w, assigning v to the 'visited' members of vertices it visits, until it comes upon the root c' of the biconnected component. The Walkup proceeds to c, where it records c' as a pertinent child biconnected component. It then continues up from c and halts when it finds a root copy v' of the vertex v currently being processed.

The Walkup for the second back edge, (v, y), starts at y again searching for the shortest path to the biconnected component root r'. It finds r' by taking the shorter path through the externally active vertex z. The Walkup then proceeds to r, where it records r' as a pertinent child biconnected component. Then, the second Walkup halts because r was already visited by the first Walkup.

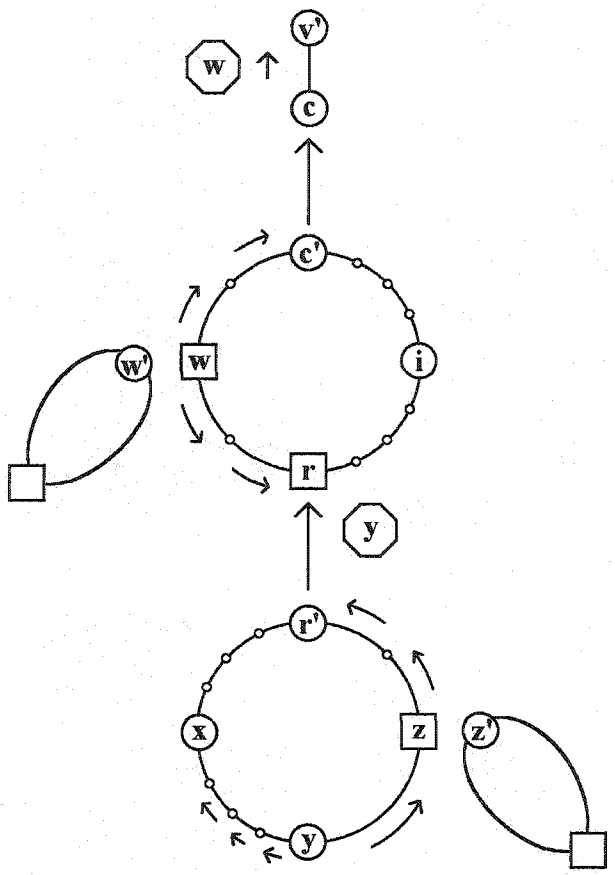


Figure 2.5: Example of Walkup Processing

2.6 Internal Activity and Inactivity

A vertex w that is not externally active may still play an active role in the embedding of back edges to the vertex v currently being processed. Vertex w could be the attachment point for a back edge to v, or it could be the attachment point for merging pertinent child biconnected components (or both).

Definition 2.6 *A vertex w is* internally active *if it is pertinent but not externally active.*

Definition 2.7 *An* internally active biconnected component *is a biconnected component that contains one or more internally active vertices and no externally active vertices.*

Definition 2.8 *A vertex is* inactive *if it is neither externally nor internally active.*

Definition 2.9 *An* inactive biconnected component *is a biconnected component that is neither externally nor internally active.*

Whether internally or externally active, a pertinent vertex w may become inactive once it becomes non-pertinent (i.e. once its back edge to v is embedded, if (v, w) exists in the input graph, and once all of its child biconnected components have been merged). Note that an externally active vertex cannot become inactive unless all of its externally active child biconnected components are pertinent since otherwise it will still have separated externally active child biconnected components at the end of step v. Also note that inactive biconnected components do not become pertinent, and if the input graph is biconnected, they do not occur.

Computationally, a biconnected component rooted by the DFS tree edge (r', c) is determined to be internally active if the lowpoint of c is equal to v. However, internal activity of a vertex w is not determined using lowpoint. According to Definitions 2.4 and 2.6, a vertex w is internally active if it is not externally active, but either its pertinentBicompList is non-empty or its adjacentTo member is equal to the current vertex v being processed, all of which are constant time tests.

2.7 Walking Down to Embed Back Edges

In the main loop of the algorithm outline in Figure 2.1, the first step runs an inner loop that invokes the Walkup procedure for each back edge from v to a descendant w in the input graph. The second step of the main loop body is to then embed all back edges from v to its descendants (unless a non-planarity condition arises).

Prior to embedding the back edges to v, each DFS tree child edge of v is the root edge of a singleton biconnected component. If v has any back edges to descendants, then one or more of its DFS tree child edges (v^c, c) in the embedding structure will have pertinent child biconnected components in the pertinentBicompList of c. For each such DFS tree child edge (v^c, c), the algorithm invokes a procedure called 'Walkdown' to embed the back edges from a root copy v^c to descendants of c, merging

pertinent child biconnected components as necessary.

For a given root copy v', the Walkdown procedure begins a sequential traversal of the external face of the biconnected component beginning with the edge indicated by the link[0] member of v'. The Walkdown merges pertinent biconnected components and embeds back edges as necessary during the traversal. These actions attach more vertices and edges to the biconnected component B rooted by v' such that the external face of B consists of more than the 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 non-pertinent externally active vertex, which is called a *stopping vertex* according to Definition 2.10. In this case, the Walkdown performs a second sequential traversal of the external face starting at v' and proceeding with the edge indicated by the link[1] member of v'. Again, the traversal proceeds to embed back edges and merge pertinent biconnected components until a stopping vertex y is encountered.

Definition 2.10 *A* stopping vertex *is a non-pertinent externally active vertex.*

As the Walkdown traverses the external faces of biconnected components, it maintains a stack of root copy vertices that must be merged when the next back edge is embedded. When a Walkdown traversal visits a vertex w, the first task is to determine whether a back edge to w must be embedded based on whether the adjacentTo member equals v. If so, then the root copies on the stack are merged with their parent copies 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. Then, the adjacentTo member of w is incremented to prevent another back edge of the form (v', w) from being embedded if w is visited again during the Walkdown.

The second task performed when visiting a vertex w is to determine whether it has any pertinent child biconnected components. If not, then w is inactive. The Walkdown obtains the successor s of w along the external face. If the adjacentTo member of s does not equal v and the biconnected component rooted by v' is externally active and contains s, then the Walkdown embeds a special 'short-circuit' edge

between v' and s along the external face such that it removes w from the external face (if the adjacentTo member of s does equal v, then w is removed from the external face when the back edge (v', s) is embedded). A short-circuit edge may not exist in the input graph, or it may form a multiple edge in $\tilde{G}$, but each short-circuit edge is specially marked for easy removal after the main loop in Figure 2.1 is finished. Short-circuit edges are added as an optimization to help achieve linear time (see Section 2.9 for details).

If, in the second task, the Walkdown determines that the pertinentBicompList of w is non-empty, 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 are in the list. Since, according to Section 2.5, the Walkup prepends the roots of internally active biconnected components and appends the roots of externally active pertinent child biconnected components, the Walkdown simply obtains w' by taking the first element of the pertinentBicompList of w.

The Walkdown pushes onto the above mentioned stack the root w' obtained from the pertinentBicompList of w, then traversal descends to the vertex w'. The direction in which to continue traversal from w' must then be selected. Due to short-circuit edges embedded by the Walkdown in prior steps of the algorithm, the immediate neighbors of w' along the external face are not inactive. If both neighbors are internally active, then either direction can be used to continue traversal. If only one external face neighbor of w' is internally active, then traversal proceeds in the direction of the internally active neighbor. If both neighbors are externally active, then traversal proceeds in the direction of a pertinent neighbor. If both neighbors are externally active and non-pertinent (i.e. both are stopping vertices), then the input graph is non-planar and the Walkdown terminates (see Section 2.8 for details on isolating a Kuratowski subgraph).

We now consider an example to help illustrate the behavior of the Walkdown. In this example, the Walkdown must embed two back edges, (v', w) and (v', y). When the first Walkdown traversal begins, the biconnected component B contains only the root edge (v', c). Since the algorithm calls Walkdown immediately after all Walkup

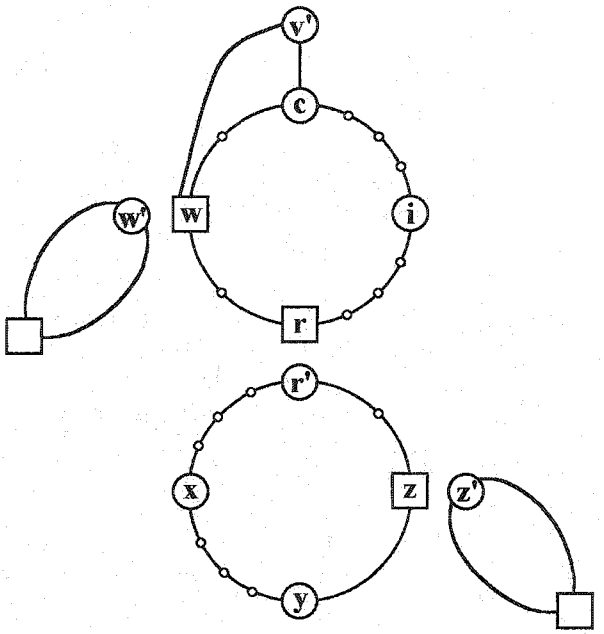


Figure 2.6: Walkdown Embeds Back Edge (v, w)

calls have been completed, Figure 2.5 from Section 2.5 exemplifies the state of our data structures at the beginning of a Walkdown on B (except that short-circuit edges are not depicted). The Walkdown proceeds from v' to c, where it finds a pertinent child biconnected component with root c'. The Walkdown descends to c', and obtains the first active vertices w and r along both external face paths extending from c'. Since both are externally active and pertinent, w is selected arbitrarily. The Walkdown merges the root copy c' with c and embeds the back edge (v', w). Figure 2.6 depicts the result of this operation (short-circuit edges are not depicted as they are not germane to correctness, though they do expedite the process of obtaining w and r).

Once the back edge to w is embedded, the Walkdown determines that it cannot proceed further beyond w because w is a stopping vertex and must remain on the external face. The Walkdown begins the second traversal at v' to walk down the opposing path (the one that does not lead to w). This path always begins with revisiting c. In this example, the second Walkdown leads to vertex r. Since the Walkup will have recorded r' as a pertinent child biconnected component of r, the Walkdown pushes r' onto the stack. However, the shorter and more natural direction of continued traversal would appear to be the path leading through z to y, but this

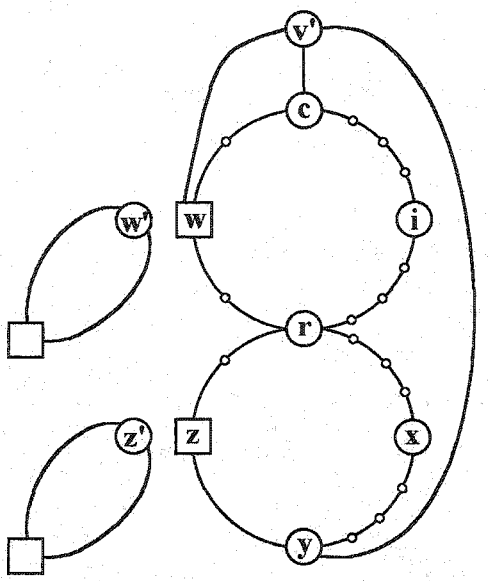


Figure 2.7: Walkdown Flips a Biconnected Component

path cannot be used because the Walkdown traversal cannot proceed beyond z since it is externally active. Instead, the Walkdown traversal selects the path through x to y because y is internally active. Note that, due to short-circuit edges (see Section 2.9), lengthy paths such as $(r', \ldots, x, \ldots, y)$ are covered by a short-circuit edge (r', y) along the external face.

Since the direction of travel used to enter r opposes the direction of travel used to exit r', the biconnected component rooted at r' must be flipped before it is merged with r. Figure 2.7 depicts the result of the flip and merge operation as well as the embedding of the back edge (v', y).

In general, a biconnected component B_c must be merged with its parent biconnected component B_p such that the path from the root of B_c to the next vertex becomes part of the proper face formed when Walkdown embeds the back edge that biconnects vertices in B_p with those in B_c. For example, in Figure 2.7 we flipped the biconnected component rooted at r' so that the path from r' through x to y would become part of the proper face formed when r and r' were merged and the back edge (v', y) was embedded.

To decide whether a biconnected component should be flipped, the Walkdown compares the direction of traversal used to enter a cut vertex r with the direction

of traversal used to exit r'. These pieces of information are pushed onto the stack when r' is pushed so that the stack has all information necessary to properly merge biconnected components when the next back edge is embedded.

Further details pertaining to flipping biconnected components appears in Section 2.10. Detailed pseudo-code for the Walkdown operation appears in Chapter 6. Detailed pseudo-code for the main algorithm loop, including the first inner loop that invokes Walkup for each back edge and the second inner loop that invokes Walkdown for each DFS tree child edge (v^c, c) with pertinent vertex c, appears in Chapter 7.

2.8 Non-Planarity and Kuratowski Subgraph Isolation

For a given biconnected component B rooted by the tree edge (v', c), if the two Walkdown traversals embed all back edges between v' and descendants of c, then B remains planar and the algorithm continues. If, however, some of the back edges to descendants of c are not embedded, then the input graph is non-planar, as shown by the proof of correctness of the Walkdown in Chapter 6.

The Walkdown may halt if it encounters two stopping vertices while trying to determine the direction of traversal from a child biconnected component root. This condition is depicted in Figure 2.8(a). Otherwise, if the Walkdown halts on a biconnected component B without embedding all back edges to descendants of a root 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 2.8(b).

Both diagrams in Figure 2.8 depict minors of the input graph. Since Figure 2.8(a) depicts a $K_{3,3}$, the input graph is clearly non-planar. However, Figure 2.8(b) appears to be planar, so it is natural to ask why the edge (v, w) was not embedded inside B, which the Walkdown is capable of doing by first 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 edge (v, w) or some aspect of the vertices embedded within B that prevents the Walkdown from embedding (v, w) inside B. Our proof of correctness exploits these two cases to identify four additional non-planarity patterns, which are also minors of the input

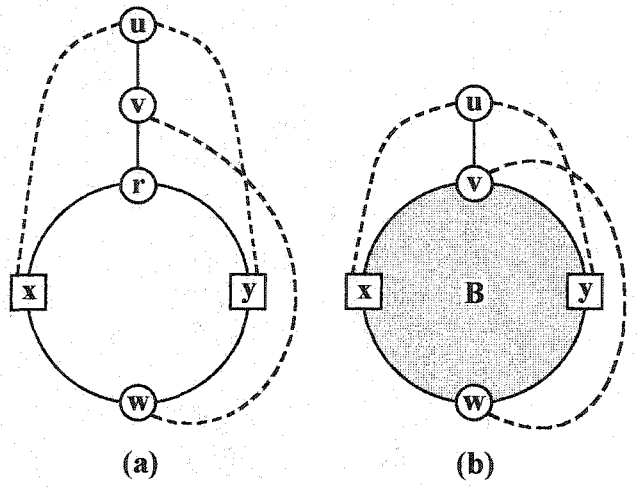


Figure 2.8: Walkdown Halting Conditions

graph. Most importantly, we prove that if none of the conditions occur that result in one of the four additional minor patterns, then the assumption of an unembedded edge (v, w) at the end of step v contradicts specific behaviors of the Walkdown. This proof relies primarily on the order of decisions made in selecting a direction of traversal from a pertinent child biconnected component root as well as the fact that a back edge to a vertex w is embedded before exploring any of its externally active pertinent child biconnected components.

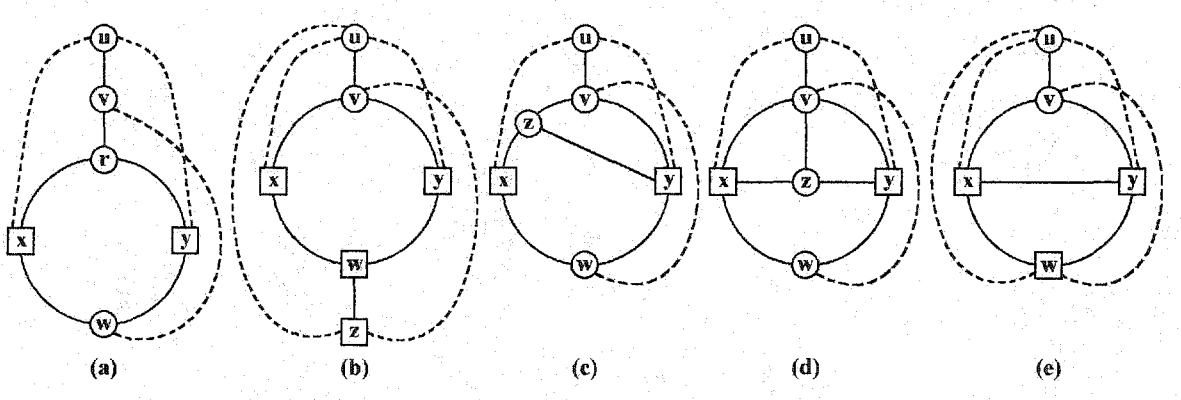


Figure 2.9: Non-planarity Minors of the Input Graph

The five non-planarity minor patterns identified in our proof of correctness are depicted in Figure 2.9. They can be used as the basis of a Kuratowski subgraph isolator. The edges in these patterns are typically representative of paths in the input graph, many of which can be found by following paths in the depth first search

tree, paths around the external faces of biconnected components, or the blocking path through B appearing in three of the patterns. Since $K_{3,3}$ is a subgraph of the patterns in Figures 2.9(b-d), the portions of the input graph indicated by some of the edges in those patterns must be omitted in order to isolate a $K_{3,3}$ homeomorph. Furthermore, the pattern in Figure 2.9(e) represents a K_5 minor in the input graph, so a few additional tests must be performed to determine whether a K_5 or $K_{3,3}$ homeomorph occurs. The details of these operations appear in Chapter 9.

2.9 Short-Circuiting Inactive Vertices

Part of our strategy of achieving linear time performance includes removing inactive vertices from the external face as they are encountered by the Walkdown. We refer to this action as *short-circuiting inactive vertices* because 'circuit' is another name for cycle, and we need to shorten the bounding cycle of a biconnected component's external face. Short-circuiting optimizes to constant time the task of obtaining the vertices necessary to choose a direction of traversal after descending to the root of a pertinent child biconnected component. The selection is performed according to the rule given in Section 2.7, which is a necessary part of the correctness of our algorithm. Figure 2.10 exemplifies a family of graphs on which our algorithm must remove inactive vertices from the external face (using a technique such as short-circuiting) in order to achieve linear time performance.

In Figure 2.10, the depth first search tree consists of a single path beginning with the topmost vertices in the order given by their Arabic numerals, proceeding to v, then clockwise through the vertices identified by Greek letters, then x and y.

At step v, all vertices not labeled with Arabic numerals are joined into a single biconnected component by embedding the back edge (v, y). Successive steps of our algorithm process the numbered vertices in descending order. For each such step $i > 0$, the biconnected component containing the tree edge $(i, i+1)$ must be merged with the child biconnected component rooted by $(i+1)'$. Before the merge can occur, the Walkdown must descend to the child biconnected component and choose a direction of traversal that leads to a pertinent vertex, with preference for an internally

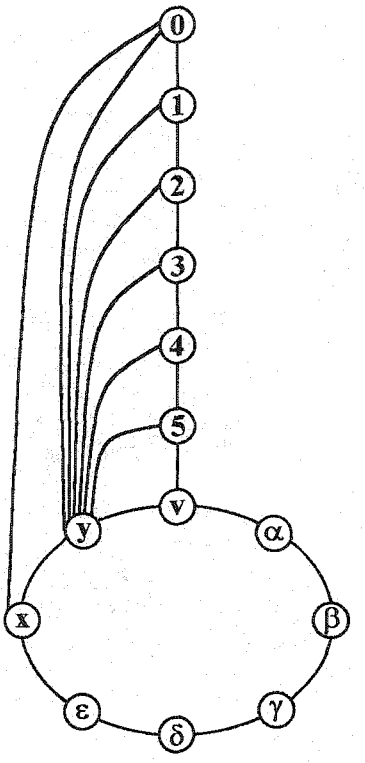


Figure 2.10: The Necessity of Short-Circuiting Inactive Vertices

active vertex. In the counterclockwise direction, the vertex y is immediately found. However, it is externally active, so the clockwise path from the root must, for reasons described below, be explored to see whether the first active vertex encountered is internally active. Without short-circuiting, the inactive vertices labeled with Greek letters would be traversed before the externally active vertex x is encountered in each step $i > 0$. Since x is a stopping vertex in each of these steps, the vertices identified by Greek letters remain on the external face after each step. If the graph is constructed such that there are $\frac{n-4}{2}$ vertices on the DFS tree path between 0 and v exclusive and between v and x exclusive, then the sum of visitations to vertices identified by Greek letters would be $(\frac{n-4}{2})^2$ were it not for short-circuiting.

The natural question at this point is why does the Walkdown examine both paths leading from a child biconnected component root if it finds a pertinent vertex along the first path examined. This, of course, is the *raison d'être* of Figure 2.11.

In Figure 2.11, the first Walkdown traversal for v merges the singleton bicon-

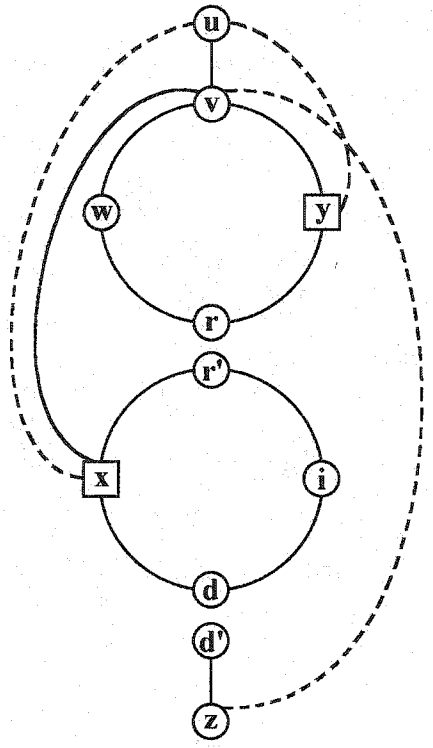


Figure 2.11: Example of Improper Selection of Traversal Direction

nected components containing y, r and w in order to embed the back edge (v, w). Then, the Walkdown continues along from w to r. It examines the pertinent child biconnected component rooted at r'. For the sake of contradiction, suppose the Walkdown selected x to be the next vertex in the traversal since it is pertinent. The result is that the biconnected component rooted at r' would not be flipped before being merged. After the Walkdown embeds the back edge (v, x), vertex x becomes a stopping vertex. The second Walkdown traversal would begin again at the root copy of v and encounter y as a stopping vertex. Thus, the back edge (v, z) would not be embedded. However, the Walkdown should not fail to embed this edge because the graph in Figure 2.11 is planar. If the Walkdown operates as describe in Section 2.7, then vertex d is selected instead of x because d is internally active.

Our planar embedding algorithm short-circuits an inactive vertex w by adding a *short circuit edge* (v', x) under the following conditions: 1) v' is the root of an externally active biconnected component B, 2) w and x are in B, and 3) x does

not have an unembedded back edge to v. The short-circuit edge appears along the external face and forms a degree three proper face with the path (v', (v', w), w, (w, x), x). A short-circuit edge may be a multiple edge with any other edge (tree edge, back edge or short-circuit edge). However, there are fewer than n short-circuit edges created during embedding because each short-circuit edge can be associated with the inactive vertex it removes from the external face. Moreover, since a short-circuit edge to x is only added under the special conditions listed above, both faces divided by a short-circuit edge are at least degree three (in the final embedding). Thus, the edge limit of Corollary 1.5 is maintained. Finally, the short-circuit edges are specially marked, so they are easily deleted at any time. Details pertaining to our method of short-circuiting appear in Chapters 3 and 4.

2.10 Flipping Biconnected Components

As described in Section 2.7, the Walkdown flips biconnected components in order to reach pertinent vertices before stopping vertices, which keeps externally active vertices on the external face while back edges are embedded. The cost of deciding whether to flip a biconnected component is dominated by the steps that decide which path to take from the root of a child biconnected component. The latter decision is reduced to constant amortized time by the use of short-circuiting and the methods for determining internal and external activity. However, the cost of flipping a biconnected component in constant amortized time has yet to be discussed.

In Chapter 1, a combinatorial planar embedding is defined to provide a clockwise ordered list of the neighbors of each vertex. To maintain a consistent vertex orientation while flipping a biconnected component would require that we invert the adjacency lists of every vertex in the biconnected component. It is easy to create graphs that would require $O(n)$ vertices to be flipped $O(n)$ times, so we cannot afford to 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. If a combinatorial planar embedding is the required result, then our algorithm marks the root edge of a biconnected component if the latter is flipped. These marked edges are used by a very simple post-processing operation to impose a consistent orientation on all of the vertices, as described in Section 2.11.

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 root copy vertex with its parent copy. When the Walkdown descends from a cut vertex w in the embedding structure to the root w' of a pertinent child biconnected component, it pushes four integers onto its stack, 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}. The link in w' indicating the edge record the Walkdown used to exit w' toward the next vertex is denoted w'_{out}. When the Walkdown finds a back edge to embed, it embeds the edge, but it also pops the stack 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.

The merge operation begins by reassigning all settings in the edge records of edges incident to w' such that the edges indicate that they are incident to 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[1 xor 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 so that a post-processing operation

described in Section 2.11 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'.

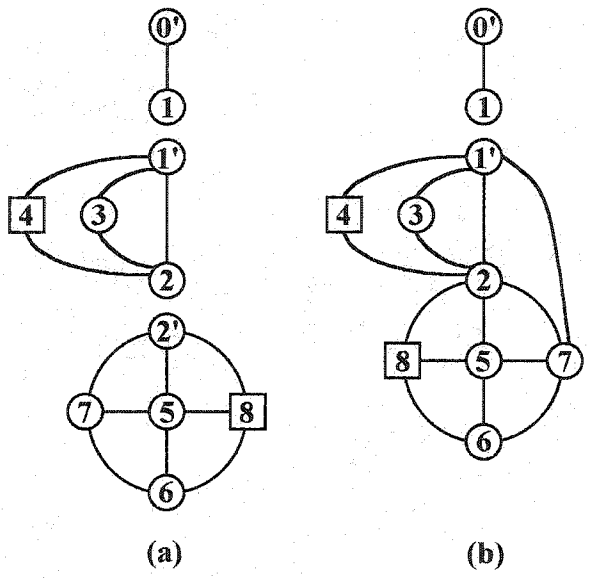


Figure 2.12: Overview of Data Structures for Flip Operation

The details of these processes are illustrated in Figures 2.12, 2.13, and 2.14. In Figure 2.12, we have an overview of the embedding of back edge (1, 7). Figure 2.12(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 2.12(b).

An elaboration of Figure 2.12(a) appears in Figure 2.13. 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.

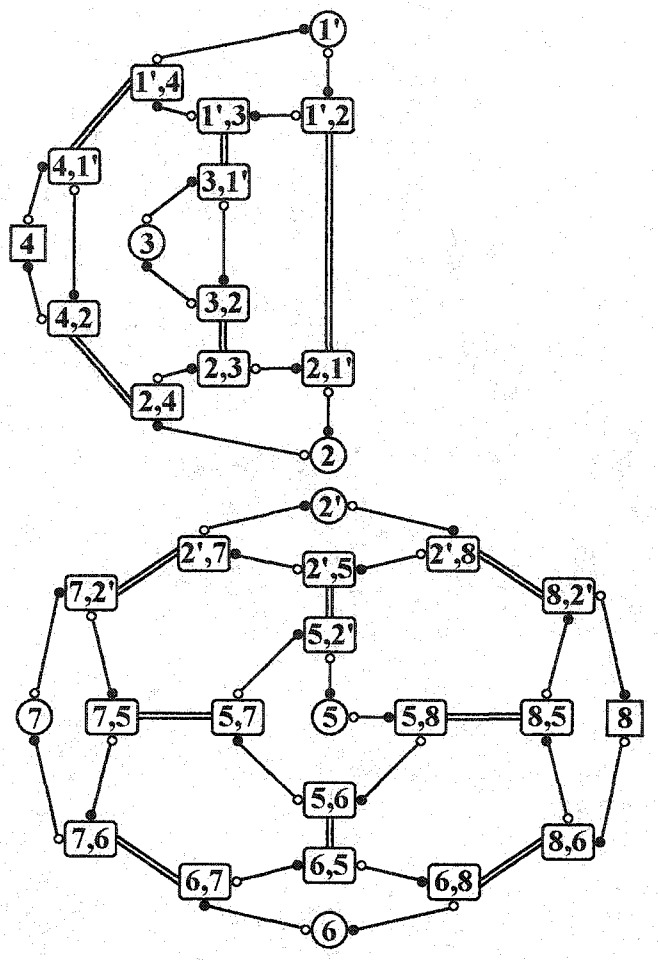


Figure 2.13: Elaboration of Data Structures Before Flip Operation

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 2.14. 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′ (the root copy vertex) 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.

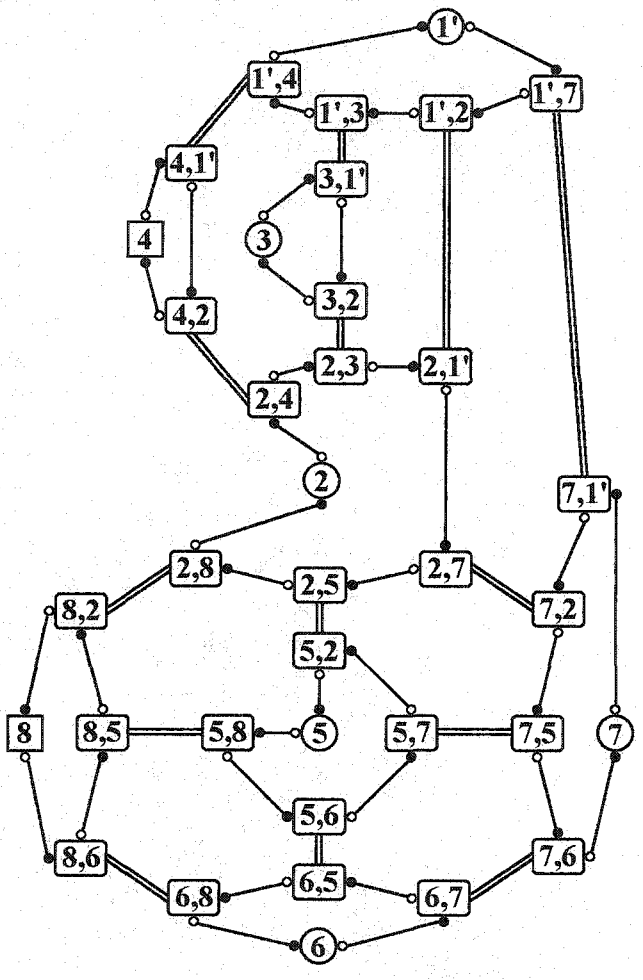


Figure 2.14: Data Structures After Flip and Back Edge Embedding

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) as described in Section 2.11.

The final change made in Figure 2.14 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 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.

The detail in Figure 2.14 makes 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 2.3). Then, our algorithms simply exit a vertex s from the opposing link. In the case of vertex 8 in Figure 2.14, 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 $1'$.

2.11 Consistent Orientation of Vertices

After the main loop described in the algorithm outline of Figure 2.1 (and detailed throughout this chapter), a simple post-processing operation can be used to impose a consistent orientation on all vertices in the embedding structure. During the merge operations described in Section 2.10, a biconnected component root edge

(w', c) is marked when the biconnected component must be flipped. The root edge is also transformed into edge (w, c) as part of the merge.

To represent the mark, we add a *sign* to each edge record that is initialized to +1. It is changed to -1 to indicate an inversion. 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. Separate biconnected components are oriented independently as described in Section 2.12. 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. Thus, a consistent orientation for all vertices in a biconnected component can be easily obtained by pre-order traversal of the portion of the depth first search tree in the biconnected component. The detailed pseudo-code for this operation appears in Chapter 8.

2.12 Handling of Separable and Disconnected Graphs

Many prior planarity algorithms operate on biconnected graphs. While it is not difficult to obtain the biconnected components using lowpoint, then embed each and reassemble the results, this means that the prior algorithms perform a portion of the work of our algorithms as a function of their preprocessing and postprocessing.

By duplicating cut vertices and making use of the lowpoint function during the embedding process, our new algorithms handle separable and disconnected graphs with virtually no modification. For a biconnected input graph, after all embedding steps and before vertex orientation, there is only one root copy vertex corresponding to the DFS tree root. If the input graph is disconnected, then there will be multiple root copy vertices representing the DFS tree root of each connected component. If any connected component is separable, then there will be root copy vertices representing the points of separation for the biconnected components within each separable connected component. Our algorithm's postprocessing is simply modified to merge these extra root copies into their parent copies (empty parent copies are created for the DFS tree roots). The detailed pseudo-code appearing in Chapter 8 includes the modifications for separable and disconnected input graphs.

3. Data Structures and Their Initialization

This chapter describes the logical components required to represent the input graph and the embedding. This chapter also describes the preprocessing of the input graph and the initialization performed on the embedding structures. The last section provides definitions that identify fundamental states of the embedding.

3.1 Notation

For the purposes of the data structures and algorithms in this dissertation, arrays are assumed to be indexed beginning with 0. The value *nil* is any invalid array index, such as -1. The number of elements of a list L is denoted $|L|$.

In the pseudo-code of this dissertation, the inputs to a procedure are indicated by the term *in* and the outputs are indicated by the term *out*. The term *this* indicates a special implicit input parameter that points to a data structure over which the procedure predominantly operates. In common object-oriented parlance, the procedure is a member of a class, and the data structure pointed to by *this* is an instance of the class. The use of this convention greatly increases the readability of the pseudo-code since *this* typically points to the embedding data structure, which is denoted $\tilde{G}$.

3.2 Preprocessing of the Input Graph

The input graph is placed in memory in adjacency list format with one structure representing each vertex and two records indicating each edge (u, v), where one edge record appears in the adjacency list of u and indicates v as a neighbor and the other edge record appears in the adjacency list of v and indicates u as a neighbor. Once the input graph is in memory in adjacency list format, it is preprocessed by a number of linear time steps. First, a depth first search is performed to assign the depth first index and depth first search parent of each vertex as well as the type of each edge record (tree edge or back edge). After the depth first search, the vertices are reordered according to their depth first indices. This facilitates future discussion

by making the label v of vertex v synonymous with the depth first index of vertex v. The final preprocessing step on the input graph is to perform the least ancestor and lowpoint calculations. All of these preprocessing calculations are required by all currently known linear time planarity algorithms [7, 30, 52].

3.3 Vertex Structures and Edge Records

This section specifies the variables used by our algorithms to represent each vertex and edge of an embedding. The same structures can be used to represent the input graph. Table 3.1 lists and describes the variables in a vertex structure. A vertex structure can either represent a parent copy or a root copy of a vertex from the input graph. Table 3.2 lists and describes the variables in edge records. An edge is represented by a pair of edge records.

For each vertex, the values of DFSParent, leastAncestor and lowpoint (see Table 3.1) are computed when preprocessing the input graph. The vertex type is assigned zero, except the bit that indicates whether the vertex structure is a parent copy or root copy. The parentLink is assigned *nil* since root copy vertices are not used until a tree edge is embedded by EmbedTreeEdge, defined in Chapter 4. The link[0] and link[1] settings are initially assigned to reference the vertex structure such that the vertex starts out with degree 0. The visited and adjacentTo members are treated as flags that are raised when equal to the index of the vertex v currently being processed, so they are initialized to the value n, which is an invalid index for zero-based arrays.

The pertinentBicompList of any descendant w of the vertex v being processed is initially empty at the beginning of step v. As a result of the calls to Walkup in step v (described in Chapter 2, Section 2.5), the pertinentBicompList of a vertex w receives the list of root copy vertices described in Table 3.1. The Walkdown calls for step v (described in Chapter 2, Section 2.7) return the pertinentBicompList of all vertices to the empty state unless the Walkdown stops with a non-planarity condition. The details described in this dissertation assume that each pertinentBicompList is a circular doubly linked list such that both prepending and appending are constant time operations.

Table 3.1: Variables in Vertex Structures

Variable	Description
DFSParent	Index of the depth first search tree parent.
leastAncestor	Minimum index of neighbors adjacent by a back edge.
lowpoint	Minimum of leastAncestor and lowpoint values of depth first search tree children.
type	Bit flags used to distinguish parent and root copy vertices. Also used by the Kuratowski subgraph isolator to classify vertices while searching for certain paths in a biconnected component.
parentLink	Indicates the parent copy vertex represented by a root copy vertex.
link[0] and link[1]	References to edge records that are used to insert the vertex structure into a circular doubly linked list with the edge records in its adjacency list. When the vertex is on the external face of a biconnected component, these links reference edge records for the two edges that join the vertex to the external face.
visited	General flag used by many algorithms (e.g., Walkup, Kuratowski subgraph isolator).
adjacentTo	For all descendants of the current vertex v, this is set before any Walkdown calls to indicate that the vertex has a back edge to v. Cleared by Walkdown once the back edge is embedded.
pertinentBicompList	For any descendant w of the current vertex v that is a cut vertex in the embedding, this member lists the roots of the child biconnected components of w that should be merged with w due to one or more unembedded back edges leading from descendant(s) of w to v.
separatedDFSChildList	For the parent copy of vertex w, this member contains a doubly linked list of DFS tree children of w that are not embedded in the biconnected component with w. The list is sorted by the lowpoint of the children of w.
repInParentList	Indicates the vertex's node in the separatedDFSChildList of its DFS parent.

The separatedDFSChildList maintained by each vertex structure is used for the efficient computation of external activity. For each vertex, the list is initialized to contain all of the depth first search tree children of the vertex, sorted by their lowpoint values. To achieve linear time, we perform the sorting indirectly. First, we bucket

Table 3.2: Variables in Edge Records

Variable	Description
neighbor	Indicates a neighbor v of the vertex w, where the adjacency list of w contains the edge record.
type	Bit flags used to distinguish tree edges, back edges, and short-circuit edges.
twinLink	For an edge record in u's adjacency list with a neighbor indicating v, the twinLink refers to the edge record in v's adjacency list with a neighbor indicating u. This member is easy to represent implicitly (e.g. by placing twin records in consecutive memory locations).
link[0] and link[1]	References to other edge records in the adjacency list or to the vertex structure containing the adjacency list.
visited	General flag used by many algorithms (e.g. Kuratowski subgraph isolator).
sign	Initially 1, changed to -1 in an edge record of edge e to indicate that the biconnected component with root edge e was flipped prior to being merged.

sort the vertices based on their lowpoint values. Since the lowpoint of a vertex v is equal to the DFI of v or one of its DFS ancestors, it suffices to create an array B of n buckets, one for each possible DFI value. A given bucket $B[i]$ should store a list of the vertices having a lowpoint of i. Once the bucket sort is finished, we iterate the buckets in ascending order, appending each vertex c to the separatedDFSChildList of its DFS parent w. The separatedDFSChildList is represented with a doubly linked list so that it can support constant time deletion.

The repInParentList member of a vertex c is set equal to the node that represents c in the separatedDFSChildList of its DFS parent w. When a biconnected component with root edge (w', c) is merged with its parent biconnected component by joining w' with its parent copy w, the DFS child c is removed from the separatedDFSChildList of w since w and c are no longer in separate biconnected components. This can be done in constant time using the repInParentList in c and the fact that the separatedDFSChildList of w is doubly linked.

Since upper bounds are known for the number of edges needed to establish the planarity or non-planarity of an input graph, an implementation is able to allocate

space for all edge records at once. The initial value for most of the variables in an edge record (see Table 3.2) should be *nil* until edges are added to the embedding by the low-level procedures EmbedTreeEdge, EmbedBackEdge, and EmbedShortCircuitEdge, which are defined in Chapter 4. The exceptions are that the type of each edge record should have all bit flags initially clear, the visited member should be 0, and the sign should be 1.

3.4 A Structure for Representing an Embedding

To complete the description of the data structures, Table 3.3 defines a structure sufficient for representing the embedding. The value of n in the embedding structure is initialized to be the number of vertices in the input graph, which is then used to allocate arrays and lists of the appropriate size as described in Table 3.3. The value of m is initialized to 0 since the embedding begins with no edges. The value of v is initialized to *nil* until the planarity algorithm begins.

Table 3.3: Variables in Embedding Structure

Variable	Description
n	The number of vertices in the embedding.
m	The number of edges embedded.
v	The current vertex whose back edges are being processed.
E	An array of $2kn$ edge records with constant $k \geq 3$.
V	An array of $2n$ vertex structures, the first n for parent copy vertices, and the latter n for root copy vertices.
BicompLists	A memory area of $O(n)$ size used to store all pertinent biconnected component lists.
DFSChildLists	A memory area of $O(n)$ size used to store all separated DFS child lists.
S	a stack used by the algorithms in this dissertation to push and pop up to $4kn$ computer words.

Based on Corollary 1.5, the array E is allocated enough edge records to accommodate either the embedding of any planar graph with n vertices or the identification of a Kuratowski subgraph in any non-planar graph with n vertices (no additional edge records are needed for short-circuit edges). The first n vertex structures in array V

are for parent copy vertices, which are meant to represent the original vertices of the input graph. The latter n vertex structures in array V are for root copy vertices. Given a cut vertex w in every path from a DFS child c to some vertex distinct from w and not a descendant of c (see Theorem 1.3), the embedding structure uses vertex structure $c + n$ in V to store a root copy w^c that represents w in the biconnected component containing c. The choice of location $c + n$ avoids allocation collisions, as shown by Lemma 3.1.

Lemma 3.1 *Given a cut vertex w separating a child biconnected component B from vertices in other biconnected components containing w, there exists only one DFS child c of w in B, so a root copy w^c can be created to represent w in B and placed at location $c + n$ in the vertex structure array V without collision with other root copies allocated in V.*

Proof. By contradiction, if two DFS children of a vertex appeared in a single child biconnected component of the vertex, then there would have to be an edge connecting the subtrees rooted by the two children. This contradicts the definition of depth first search tree in which the only non-tree edges are back edges connecting an ancestor and descendant. Moreover, a DFS child c will only be in one child biconnected component since it only has one DFS parent w. Thus, the index c can be used to uniquely identify the child biconnected component of w that contains c, and we add n to determine the location of its root copy vertex because the first n locations of V are, by definition, allocated for parent copy vertices. □

The BicompLists and DFSChildLists members are simple memory areas used to store the nodes of every pertinentBicompList and separatedDFSChildList, respectively. The space is sufficient because each pertinentBicompList contains root copy vertices that differ from those in every other pertinentBicompList and no separatedDFSChildList ever contains vertices that are in another separatedDFSChildList.

The stack S, initially empty, contains enough computer words to accommodate pushing one pair of computer words per edge record in E. Elementary algorithms such as depth first search on the input graph can be written to push a vertex plus a pointer

to an edge record leading from that vertex to a neighbor. The stack S is also used as temporary space by the procedures MergeBicomps and Walkdown in Chapter 6, the procedure OrientVertices in Chapter 8 and procedures of the Kuratowski subgraph isolator in Chapter 9.

Finally, note that since edge records are stored in array E while vertex structures are stored in V, the implementation and use of link[0] and link[1] in edge records requires further consideration because we insert vertex structures into doubly linked circular lists along with the edge records in their adjacency lists. In every case except link[0] and link[1] in edge records, it is clear whether a structure or record variable refers to E or V. For example, the link[0] and link[1] members of a vertex structure always refer to E, so these variables can be implemented as integers that hold array indices. However, link[0] and link[1] in an edge record could indicate another edge record in E, or the inserted vertex structure from V.

One simple solution is to create extra edge records in E to store the link[0] and link[1] members of vertex structures, and to use the type and neighbor fields of those edge records to bind them to vertex structures in V. More elegant solutions are possible depending on the sophistication of the implementation language. For example, in object oriented languages, the edge records and vertex structures can inherit from a common base class containing the type as well as link[0] and link[1], implemented as physical memory pointers. Upon traversing a link to arrive at a new edge record or vertex structure x, the type would be used to determine whether x is in E or V. Though other solutions can be created based on minor modifications to the embedding structure presented in this section, the above methods demonstrate that the edge record links can be implemented in constant space and traversed in constant time. In the pseudo-code of this dissertation, the matter is treated as an implementation detail.

3.5 The Initialization Procedure

The prior sections of this chapter lead to an initialization algorithm for the data structures described. Figure 3.1 encapsulates this initialization algorithm in

Figure 3.1: The Initialization Procedure

Procedure: Initialize
in: Graph G with at most kn edges
out: Embedding Structure $\tilde{G}$

(1) Create DFS tree in G
(2) Reorder vertices in G according to DFI
(3) Compute leastAncestor and lowpoint in G

(4) Allocate embedding structure $\tilde{G}$ and its subordinate structures
(5) Set n in $\tilde{G}$ equal to the number of vertices n in G
(6) Assign *nil* to the 'current vertex' member v in $\tilde{G}$

(7) Initialize vertex structures in array V of embedding structure $\tilde{G}$
(8) Copy DFSParent, leastAncestor and lowpoint of each vertex G
to vertex structures in array V of $\tilde{G}$

(9) Set m in $\tilde{G}$ equal to 0
(10) Initialize edge records in array E of $\tilde{G}$

(11) Clear the BicompLists memory array in $\tilde{G}$
(12) Clear the DFSChildLists memory array in $\tilde{G}$
(13) Create sorted separated DFS child lists
for all vertices with a DFS parent

(14) Initialize the stack S in $\tilde{G}$ to be empty

(15) return $\tilde{G}$

a procedure named Initialize, which is expressed with a pseudo-code notation used throughout this dissertation. Lemma 3.2 states that the Initialize procedure achieves $O(n)$ performance.

Lemma 3.2 *The procedure Initialize achieves $O(n)$ performance.*

Proof. The input graph is assumed to have $O(n)$ edges. Lines 1 to 3 call upon well-known algorithms [53] that achieve linear time performance. Line 4 is typically constant time, but linear time in languages that insist on clearing the bits of allocated memory space with a simple loop. Lines 5, 6, 9 and 14 are simple constant time

assignments, and Lines 7, 8, 10, 11 and 12 require simple linear time loops. Line 13 achieves linear time by the method described in Section 3.3. Line 15 is constant time since a pointer to the initialized embedding structure is returned. □

3.6 Fundamental Embedding Data Structure Definitions

Based on the vertex structure, edge record and embedding structure specifications appearing in Tables 3.1, 3.2, and 3.3, it is possible to define a root copy, an adjacency list, and the representation of an edge according to the needs of the algorithms in this dissertation. For the sake of future reference, these are presented formally as Definitions 3.1, 3.2, 3.3, and 3.4.

Definition 3.1 *A root copy w^c of a vertex w is a vertex structure at position $c+n$ in the vertex array V of the embedding whose type field indicates that it is a root copy and whose parentLink indicates the vertex structure for w. The root copy w^c represents the vertex w in the biconnected component containing the DFS child c of w until the first back edge from c or one of its descendants to an ancestor of w is embedded, biconnecting c with ancestors of w.*

Definition 3.2 *The adjacency list of a vertex structure w is defined to be the list of zero or more edge records that can be obtained by iterating the successive link[p] members starting with the vertex structure w and ending when the iteration returns to w, where p can be either zero or one. The neighbor member of each edge record indicates a vertex structure in use by the embedding structure $\tilde{G}$.*

Definition 3.3 *A tree edge or back edge between two vertex structures x and y is represented by two edge records e_x and e_y as follows:*

1. *One unit of the value of m in the embedding structure represents the edge corresponding to the edge records e_x and e_y.*

2. *In accordance with Definition 3.2, e_x appears in the adjacency list of x and e_y appears in the adjacency list of y.*

3. *The neighbor of e_x indicates y, and the neighbor of e_y indicates x.*

4. *The twinLink of e_x indicates e_y, and the twinLink of e_y indicates e_x.*

5. *If x is a root copy or parent copy of the DFSParent of y, then the type of e_x and e_y is tree edge. Otherwise, the type is back edge.*

6. *The value of visited and sign are not relevant to edge representation, so they are not altered when a pair of edge records is used to represent an edge, and they can be changed as needed for other purposes.*

Definition 3.4 *A short-circuit edge between two vertex structures x and y is represented exactly as a back edge (see Definition 3.3) except that the type members of the edge records are set to indicate a short-circuit edge. A short-circuit edge (x, y) cannot exist except when it forms a degree three proper face in which the third vertex i is both inactive and has link[0] and link[1] members that indicate edges incident to x and y.*

The initial state of the embedding $\tilde{G}$ prior to the embedding of any edges results from the execution of procedure Initialize given in Section 3.5. This state is characterized by Definition 3.5.

Definition 3.5 *Given an input graph G, the* initial state *of the embedding structure $\tilde{G}$ is as follows:*

1. *The value of n is equal to the number of vertices in G.*

2. *The value of m is equal to zero (no edges have been embedded).*

3. *The value of v is n (the main back edge embedding loop has not begun).*

4. *The members E, V, BicompLists, DFSChildLists and S have the sizes indicated by Table 3.3.*

5. *The members BicompLists and S are empty.*

6. *Each edge record of the array E has the members defined by Table 3.2. The neighbor, link[0], link[1] and twinLink members are equal to nil, the type and visited members are equal to zero, and the sign is equal to one.*

7. *Each vertex structure in the array V has the members defined by Table 3.1. The parent copy vertex structures occupy the first n locations of V, and the root copy vertex structures occupy the second n locations of V.*

8. *For each vertex structure, the adjacentTo and visited members are equal to n (one greater than the index of the highest numbered vertex in G).*

9. *For each vertex structure, the link[0] and link[1] members indicate the vertex structure such that each vertex has degree zero.*

10. *For each vertex structure, the parentLink is equal to nil.*

11. *For each vertex structure, the type member is equal to zero for a parent copy or one for a root copy.*

12. *For each parent copy vertex structure, the members DFSParent, leastAncestor and lowpoint are based on the corresponding values for vertices in G, which are defined in Table 3.1 and calculated by preprocessing G in the manner described in Section 3.2. These three members are not used in root copies.*

13. *For each parent copy vertex structure, the pertinentBicompList is empty. This member is not used in root copies.*

14. *For each parent copy vertex structure, the separatedDFSChildList contains the indices of the DFS children of the vertex in ascending order by the lowpoint values of the DFS children. This member is not used in root copies.*

15. *For each parent copy vertex structure, the repInParentList indicates the node in the DFS parent's separatedDFSChildList that contains the index of the vertex.*

16. *The nodes of every separatedDFSChildList are stored in DFSChildLists.*

According to the high-level pseudo-code given in Figure 2.1, Lines 1 to 3 perform the tasks associated with procedure Initialize such that the embedding structure is set to the initial state defined in Definition 3.5. This is followed by embedding the DFS tree edges in Line 4 (before embedding any back edges). The embedding of the DFS tree edges results in the second fundamental state of the embedding $\tilde{G}$, which is described in Definition 3.6.

Definition 3.6 *Immediately after embedding all DFS tree edges of the input graph G and prior to the first iteration of the main back edge embedding loop, the* tree edge state *of the embedding structure $\tilde{G}$ is as in the initial state of Definition 3.5 with two exceptions. First, the value of m becomes $n - C$ where n is the number of vertices in G and C is the number of connected components in G. Second, each tree edge (p, c) with DFS parent p and DFS child c is embedded (in accordance with Definition 3.3) incident to the root copy p^c and the parent copy of c.*

In the main loop starting on Line 5 of the high-level pseudo-code given in Figure 2.1, the vertices are processed in descending order based on depth first index by embedding the back edges from each vertex v to its DFS descendants (either until all have been embedded or until sufficient conditions are found to permit the conclusion that G is non-planar). As each edge is embedded, the algorithm maintains Properties 3.1, 3.2, 3.3, 3.4, 3.5, 3.6 and 3.7, which lead to the defined state given in Definition 3.7 at the beginning of each step v of the main back edge embedding loop.

Property 3.1 Collection of Planar Biconnected Components: *The embedding $\tilde{G}$ consists of a collection of biconnected components containing tree edges, back edges and short-circuit edges embedded according to Definitions 3.3 and 3.4. Each biconnected component B represents a planar subgraph of the input graph G that is distinct from the subgraphs represented by all other biconnected components in $\tilde{G}$ except for a single root copy representing the vertex of least DFI in B.*

Property 3.2 Active Vertices are Maintained on External Face: *Every internally and externally active parent copy vertex appears on the external face of the biconnected*

component containing the parent copy vertex (root copies are always on the external face).

Property 3.3 External Face Edges are Maintained by External Face Vertices: *Every parent copy and root copy vertex structure w on the external face of a biconnected component B has its link[0] and link[1] members set to indicate the edge(s) that join w on the external face. If B contains only one edge, then both links indicate the same edge. Otherwise, the links indicate distinct edges.*

Property 3.4 Clockwise Orientation of Vertex Adjacency Lists: *For each vertex structure w in an embedding, consider the nearest root copy r' in the embedding, if one exists, which could be obtained by traversing zero or more distinct embedded tree edges. Let parity be 0 unless there exists such an r' and the number of tree edges traversed to obtain r' that have a sign of -1 is odd, in which case let parity be 1. A clockwise list of the neighbors of w is obtained by traversing the link[parity] members of the adjacency list of w. Note that for a root copy r', the parity is 0 since no tree edges are traversed to reach it.*

Property 3.5 Unnecessary Root Copies are Eliminated: *For every tree edge (p, c) for which a back edge has been embedded between an ancestor of p and c or a descendant of c, the root copy p^c has been eliminated and all of its incident edges have been made incident to the parent copy of p.*

Property 3.6 Dynamic External Activity: *For any root copy p^c created in Definition 3.6, if and only if p^c has been eliminated (due to maintaining Property 3.5), then the indicator of c no longer appears in the separatedDFSChildList of p. Thus, the external activity of a parent copy vertex structure p is only dependent upon its leastAncestor member and the lowpoint members of those DFS children whose parent copy vertex structures are not in the same biconnected component as p.*

Property 3.7 Edge Count: *The value of m in the embedding structure is equal to the sum of the number of tree edges, back edges and short-circuit edges (if any) in the embedding.*

Definition 3.7 *The embedding is in the* vertex invariant state *if it conforms to the tree edge state of Definition 3.6 except as follows:*

1. *The value of v in the embedding structure is equal to the current step value (which indicates the vertex v whose back edges to descendants are to be embedded in the current iteration of the main loop body).*
2. *The visited member of each vertex structure is greater than v (which is logically equivalent to being equal to n).*
3. *The embedding contains short-circuit edges and all back edges from G for which both endpoints are vertices with a DFI greater than or equal to $v + 1$.*
4. *All embedded edges collectively conform to Properties 3.1, 3.2, 3.3, and 3.4.*
5. *The embedding has Properties 3.5, 3.6, and 3.7.*

As described in Chapter 2, the two principal activities involved in embedding the back edges from v to its descendants are 1) Walking up to set adjacentTo flags and create pertinentBicompList entries, and 2) Walking down to embed back edges, which clears adjacentTo flags and deletes pertinentBicompList entries as the back edges are embedded and the biconnected component roots are merged with their parent copies. The handling of the adjacentTo and pertinentBicompList members of vertices is captured in Property 3.8. The combination of Properties 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, and 3.8 lead to a defined state of the embedding given in Definition 3.8.

Property 3.8 Dynamic Pertinence: *For any back edge (v, d) in the input graph G from vertex v to a descendant d, let v^c denote a root copy of v in the biconnected component containing the DFS child c of v that is an ancestor of w.*
If the back edge (v, d) has not been embedded, then the following are true:

1. *The adjacentTo member of d equals v.*
2. *For any separation ancestor w of d in $\tilde{G}$ with DFS child a that is ancestor to or equal to d, an indicator of the root copy w^a appears in the pertinentBicompList*

of w. The indicators of roots of internally active biconnected components appear earlier in the pertinentBicompList of w than the indicators of roots of externally active biconnected components.

If the back edge (v, d) has been embedded, then the following are true:

1. *The adjacentTo member of d equals n.*
2. *For every ancestor w of d in $\tilde{G}$ with DFS child a equal to or ancestor to d and with v a DFS ancestor of w, vertex w is not a separation ancestor of d and the pertinentBicompList of w does not contain an indicator of the root copy vertex structure w^a (which is no longer in use within $\tilde{G}$).*

Definition 3.8 *The embedding is in the* embedding invariant state *if it conforms to the vertex invariant state of Definition 3.7 with four exceptions:*

1. *The visited members of vertex structures are either equal to or greater than v.*
2. *The stack S may be non-empty.*
3. *The embedding has Property 3.8 (with the nodes of every pertinentBicompList being stored in BicompLists).*
4. *The embedding may contain back edges and short-circuit edges between v and its descendants, with Properties 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7 and 3.8 maintained.*

Definition 3.9 *The embedding is in the* oriented embedding invariant state *if it conforms to the embedding invariant state of Definition 3.8, there are no short-circuit edges, and the sign of all edge records is 1.*

4. Low Level Operations

This chapter describes a plethora of simple, low-level procedures called by the planarity testing and embedding algorithms described in Chapters 5, 6, 7 and 8. A few of the procedures are also used by the Kuratowski subgraph isolator in Chapter 9.

4.1 Embedding a DFS Tree Edge

The procedure EmbedTreeEdge defined in Figure 4.1 embeds a DFS tree edge from the input graph. Each tree edge (p, c) is embedded as a singleton biconnected component consisting of a root copy vertex p', the parent copy of c, and two edge records to represent the edge (p, c).

The correctness of EmbedTreeEdge is implied in Lemma 4.2 by virtue of the

Figure 4.1: The Procedure for Embedding a DFS Tree Edge

Procedure: EmbedTreeEdge
this: Embedding Structure $\tilde{G}$
in: DFS Parent vertex p
in: DFS Child vertex c
out: Embedding Structure $\tilde{G}$ containing a root copy p'
of vertex p and edge records representing the tree edge
between p' and c

(1) Let $c + n$ be the location of a root copy p'
(2) Set the parentLink of p' equal to p

(3) Let $2m$ and $2m + 1$ be the locations in E of new edge records
e_p and e_c (the twinLink between e_p and e_c is implicit)
in their consecutive locations)
(4) Insert e_p as the sole edge record in the adjacency list of p'
(5) Insert e_c as the sole edge record in the adjacency list of c
(6) Set the neighbor of e_p to c and the neighbor of e_c to p'
(7) Set the type of e_p and e_c to 'Tree Edge'

(8) Increment m

collective operation of all invocations of EmbedTreeEdge indicated in Precondition 4.1. The running time for EmbedTreeEdge is stated in Lemma 4.3.

Precondition 4.1 *Beginning with an embedding structure $\tilde{G}$ in the initial state (Definition 3.5), the procedure PlanarityTest described in Chapter 7 invokes EmbedTreeEdge once per DFS tree edge in the input graph G.*

Lemma 4.2 *As a result of the invocations of EmbedTreeEdge described in Precondition 4.1, the embedding structure $\tilde{G}$ is transformed to the tree edge state (Definition 3.6).*

Proof. Since p' has the proper type due to Definition 3.5, Lines 1 and 2 of EmbedTreeEdge complete the allocation of p' as a root copy according to Definition 3.1. Lines 3 to 8 of EmbedTreeEdge complete the representation of a DFS tree edge according to Definition 3.3. Precondition 4.1 completes the proof. □

Lemma 4.3 *The procedure EmbedTreeEdge runs in $O(1)$ time.*

4.2 Embedding a Back Edge

The procedure EmbedBackEdge defined in Figure 4.2 embeds a back edge between a root copy v' of the vertex v currently being processed and a descendant w of v that, in the input graph, is adjacent to v by a back edge. The back edge is embedded such that it appears on the external face of the biconnected component rooted by v'.

The procedure EmbedBackEdge is called by the Walkdown procedure in Chapter 6, which provides two values, v'_{out} and w_{in}, that indicate the edge record used to exit v' and enter w, respectively. These values help to indicate the path used by the Walkdown to travel from v' to w, and they also indicate where the new edge records for (v', w) should be inserted into the adjacency lists of v' and w.

The correctness of procedure EmbedBackEdge, based on Precondition 4.4, is established in Lemma 4.5. The running time for procedure EmbedBackEdge is stated in Lemma 4.6.

Figure 4.2: The Procedure for Embedding a Back Edge

Procedure: EmbedBackEdge
this: Embedding Structure $\tilde{G}$
in: Root copy v' of vertex being processed
in: Indicator v'_{out} (0 or 1) of the edge record used to exit v'
in: Descendant vertex w
in: Indicator w_{in} (0 or 1) of the edge record used to enter w
out: Embedding Structure $\tilde{G}$ modified to include the back edge (v', w)

(1) Let $2m$ and $2m+1$ be the locations in E of new edge records
    $e_{v'}$ and e_w (the twinLink between $e_{v'}$ and e_w is implicit)
    in their consecutive locations)
(2) Insert $e_{v'}$ between the vertex structure for v' and the link$[v'_{out}]$
    edge record in the adjacency list of v'
(3) Insert e_w between the vertex structure for w and the link$[w_{in}]$
    edge record in the adjacency list of w
(4) Set the neighbor of $e_{v'}$ to w and the neighbor of e_w to v'
(5) Set the type of $e_{v'}$ and e_w to 'Back Edge'
(6) Increment m

Precondition 4.4 *With respect to embedding a back edge (v, w), the Walkdown procedure described in Chapter 6 is operating on a biconnected component with root v^c, where c is an ancestor of w. Starting within an embedding in the embedding invariant state (Definition 3.8), the Walkdown performs zero or more biconnected component merge operations such that, after adding (v^c, w), the vertex structures v^c and w are guaranteed to be in the same biconnected component. The Walkdown invokes EmbedBackEdge with v'_{out} and w_{in} such that if the edge records for (v^c, w) are indicated by link$[v'_{out}]$ of v^c and link$[w_{in}]$ of w, then (v^c, w) is embedded along the external face of the biconnected component rooted by v^c and the resulting embedding is returned to the embedding invariant state (Definition 3.8).*

Lemma 4.5 *Given an invocation subject to Precondition 4.4, the EmbedBackEdge procedure embeds the back edge (v', w) along the external face of the biconnected component containing v' and w such that the edge records are indicated by link$[v'_{out}]$ in v' and link$[w_{in}]$ in w. The resulting embedding conforms to Definition 3.8.*

Proof. Lines 1 to 6 add a back edge conforming to Definition 3.3. The remainder of the proof follows by assumption of Precondition 4.4. □

Lemma 4.6 *The EmbedBackEdge procedure runs in $O(1)$ time.*

4.3 Embedding a Short-Circuit Edge

The procedure EmbedShortCircuitEdge defined in Figure 4.3 embeds a short-circuit edge between a root copy v' of the vertex v currently being processed and a descendant w of v. The purpose of a call to this procedure is to embed an extra edge such that the procedure GetNextVertexOnExternalFace (see Section 4.5) skips over inactive vertices between w and the root of the biconnected component containing w. This is helps our algorithms to achieve worst case $O(n)$ performance.

Figure 4.3: The Procedure for Embedding a Short-Circuit Edge

Procedure: EmbedShortCircuitEdge
this: Embedding Structure $\tilde{G}$
in: Root copy v' of vertex being processed
in: Indicator v'_{out} (0 or 1) of the edge record used to exit v'
in: Descendant vertex w
in: Indicator w_{in} (0 or 1) of the edge record used to enter w
out: Embedding Structure $\tilde{G}$ modified to include the short-circuit edge (v', w)

(1) EmbedBackEdge(v', v'_{out}, w, w_{in})
(2) Change type fields in edge records for new edge to 'Short Circuit Edge'

The correctness of procedure EmbedShortCircuitEdge, based on Precondition 4.7, is established in Lemma 4.8. The running time for procedure EmbedShortCircuitEdge is stated in Lemma 4.9.

Precondition 4.7 *Starting within an embedding in the embedding invariant state (Definition 3.8), the Walkdown procedure described in Chapter 6 invokes EmbedShortCircuitEdge with v'_{out} and w_{in} to add edge (v', w) such that if the edge records for*

(v', w) are indicated by link[v'_{out}] of v' and link[w_{in}] of w, then (v', w) creates a degree three proper face with the path (v', (v', i), i, (i, w), w), where i is both inactive and has link[0] and link[1] members that indicate edges incident to v' and w. The Walkdown invokes EmbedShortCircuitEdge only if v' and w are in the same biconnected component, only if that biconnected component is externally active, and only if $(v, w) \notin E(G)$ or $(v, w) \in E(\tilde{G})$ (i.e. only if there is no unembedded back edge (v, w)).

Lemma 4.8 *Given an invocation subject to Precondition 4.7, the EmbedShortCircuitEdge procedure embeds the short-circuit edge (v', w) along the external face of the biconnected component containing v' and w such that the edge records are indicated by link[v'_{out}] in v' and link[w_{in}] in w. The resulting embedding conforms to Definition 3.8.*

Proof. Line 1 invokes EmbedBackEdge since Precondition 4.7 satisfies Precondition 4.4. The resulting embedding conforms to Definition 3.8 by Lemma 4.5, and Line 2 does not change this. Line 2 changes the type of the edge records so that (v', w) conforms to Definition 3.4. □

Lemma 4.9 *The EmbedShortCircuitEdge procedure runs in $O(1)$ time.*

4.4 Activity Status of Vertices

The procedure VertexActiveStatus defined in Figure 4.4 determines whether a vertex is externally active, internally active or inactive. These terms are defined in Chapter 2 (and are not applicable to root copy vertices).

Lemma 4.11 establishes the correctness of the VertexActiveStatus procedure. Note that part of the proof relies on Lemma 4.10, which describes the condition for testing the external activity of a biconnected component. Lemma 4.12 states the running time of the VertexActiveStatus procedure.

Figure 4.4: Procedure for Determining Vertex Activity Status

Procedure: VertexActiveStatus
this: Embedding Structure $\tilde{G}$
in: Vertex w, a descendant of v
out: Indication that w is externally active, internally active or inactive

(1) if the leastAncestor member of w is less than v
(2) return 'Externally Active'
(3) Let c be the first element of the separatedDFSChildList of w
(4) if $c \neq nil$ and the lowpoint of c is less than v
(5) return 'Externally Active'

(6) if the adjacentTo member of w equals v or |pertinentBicompList of w| > 0
(7) return 'Internally Active'

(8) return 'Inactive'

Lemma 4.10 *A biconnected component B rooted by tree edge (w', c) is externally active if and only if the lowpoint of c is less than the index of the vertex v currently being processed.*

Proof. If the lowpoint of c is less than v, then either c is adjacent to an ancestor of v or c has a descendant that is adjacent to an ancestor of v. In the first case, c is an externally active vertex in B. In the second case, the descendant is either in B or a descendant d of a vertex b in B. If the former, then B contains an externally active vertex. If the latter, then d establishes the external activity of b as follows. Form a sequence S from the cut vertices along the DFS path from d to b, which includes b. Vertex d establishes the external activity of the first vertex in S, and each element of S establishes the external activity of its successor in S. Thus, b is an externally active vertex, and B is externally active.

Inversely, if the lowpoint of c is not less than v, then neither c nor any of its descendants have any back edges to an ancestor of v. Therefore, there is no descendant of any vertex in B that is adjacent to an ancestor of v and whose DFS ancestor chain could be followed in the manner described above to establish the external activity of

a vertex in B. Furthermore, no vertices in B are adjacent to an ancestor of v because they are also descendants of c. Thus, B contains no externally active vertices. □

Lemma 4.11 *Given an embedding $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a parent copy vertex w, the procedure VertexActiveStatus determines whether w is externally active, internally active or inactive.*

Proof. Lines 1 and 2 examine only the leastAncestor field of vertex w to determine if w has a back edge to an ancestor of v. If this were insufficient, then w would have a back edge to an ancestor of v despite the failure of the comparison in Line 1, which contradicts the definition of leastAncestor in Definition 3.8 (and hence Table 3.1).

Lines 3 to 5 examine only the lowpoint of the first separated DFS child c, if it exists, to test whether there exists a biconnected component with root copy w' that contains one or more externally active vertices. If there is no separated DFS child, then there are no child biconnected components containing root copies of w. If there are one or more separated DFS children, then only the first element c of the separatedDFSChildList must be considered since the list is sorted by lowpoint according to Definition 3.8. According to Lemma 4.10, the biconnected component with root edge (w', c) contains one or more externally active vertices if and only if the lowpoint of c is less than v. Thus, Lines 1 to 5 determine whether a vertex is externally active according to Definition 2.1.

According to Definition 3.8, the adjacentTo flag is raised in vertex w if there is a back edge (v, w) in the input graph G that has not been embedded in $\tilde{G}$. Also according to Definition 3.8, the root vertex w^c of a biconnected component is added to the pertinentBicompList of a vertex w when there is a back edge (v, d) in G but not $\tilde{G}$, where d is equal to or a descendant of c and c is a child of w. Thus, since the tests of adjacentTo and pertinentBicompList in Line 6 occur after the external activity test in Lines 1 to 5, VertexActiveStatus determines whether a vertex is internally active according to Definition 2.6.

Finally, Line 8 cannot return a result of inactive for w unless both the tests for external and internal activity have failed. Thus, VertexActiveStatus determines

whether a vertex is inactive according to Definition 2.8. □

Lemma 4.12 *The procedure VertexActiveStatus runs in $O(1)$ time.*

4.5 Traversing the External Face

The procedure GetNextVertexOnExternalFace defined in Figure 4.5 is the routine used to traverse the external faces of biconnected components in the embedding $\tilde{G}$. The procedure receives a vertex w and an indicator w_{in} (0 or 1) of the link in w's vertex structure that leads to w's predecessor. The procedure returns the successor s as well as an indicator s_{in} (0 or 1) of the link in s's vertex structure that leads to w.

The notable exception occurs if s is degree one, since both link[0] and link[1] indicate an edge record that leads back to w. However, this case only occurs when w and s are the only two vertices in a biconnected component containing a single tree edge. Since w and s are assumed to have the same orientation when embedded by EmbedTreeEdge, s_{in} is set equal to w_{in} in this case. This allows s and s_{in} to be used by other procedures such as EmbedBackEdge in Section 4.2. If s is the endpoint of a back edge, then it is necessary for s_{in} to indicate whether link[0] or link[1] should be changed by EmbedBackEdge.

The correctness of procedure GetNextVertexOnExternalFace is stated in Lemma 4.13. The running time for procedure GetNextVertexOnExternalFace is stated in Lemma 4.14.

Lemma 4.13 *Given an embedding $\tilde{G}$ in the embedding invariant state (Definition 3.8), a root copy or parent copy vertex w and a 0 or 1 value w_{in} indicating the edge record link used to enter w, the procedure GetNextVertexOnExternalFace returns the successor s and the 0 or 1 value s_{in} indicating the edge record link used to enter s.*

Lemma 4.14 *The GetNextVertexOnExternalFace procedure runs in $O(1)$ time.*

Figure 4.5: The Procedure Used to Traverse the External Face

Procedure: GetNextVertexOnExternalFace
this: Embedding Structure $\tilde{G}$
in: A vertex w
in: An indicator w_{in} (0 or 1) of the link in w that refers to the edge record used to enter w (a subsequent call exits via the opposing link)
out: The successor s of w and the indicator s_{in} of the edge record used to enter s

(1) $w_{out} \leftarrow 1$ xor w_{in}
(2) $e_w \leftarrow$ the link[w_{out}] edge record of vertex w

(3) $e_s \leftarrow$ the twinLink of edge record e_w
(4) if link[0] in edge record e_s indicates a vertex structure
(5) $\quad s \leftarrow$ link[0] in edge record e_s
(6) else $s \leftarrow$ link[1] in edge record e_s

(7) if link[0] and link[1] in vertex s indicate the same edge record
(8) $\quad s_{in} \leftarrow w_{in}$
(9) else if link[0] in vertex s is equal to e_s
(10) $\quad s_{in} \leftarrow 0$
(11) else $s_{in} \leftarrow 1$

(12) return (s, s_{in})

4.6 Inverting the Orientation of a Vertex

The procedure InvertVertex defined in Figure 4.6 reverses the roles of link[0] and link[1] in the vertex structure of a vertex w as well as all edge records in w's adjacency list. This procedure is called from procedure MergeBicomps defined in Chapter 6 and from procedure OrientVertices defined in Chapter 8.

The correctness of procedure InvertVertex is stated by Lemma 4.15. The run-time analysis for procedure InvertVertex is stated in Lemma 4.16.

Lemma 4.15 *Given a vertex structure $w \in \tilde{G}$ with an adjacency list that conforms to Definition 3.2, the InvertVertex procedure swaps the link[0] and link[1] members of the vertex structure w and the edge records in the adjacency list of w, after which the adjacency list of w conforms to Definition 3.2.*

Figure 4.6: The Procedure InvertVertex

Procedure: InvertVertex
this: Embedding Structure $\tilde{G}$
in: A vertex structure w
out: Embedding Structure $\tilde{G}$ modified by swapping link[0] and link[1] in vertex structure w structure and each edge record in its adjacency list

(1) $e \leftarrow$ link[0] edge record of w
(2) Swap link[0] and link[1] in w
(3) while $e \neq w$
(4) $\quad e_{next} \leftarrow$ link[0] edge record of e
(5) $\quad$ Swap link[0] and link[1] in e
(6) $\quad e \leftarrow e_{next}$

Lemma 4.16 *The InvertVertex procedure runs in $O(d)$ time, where d is the degree of the vertex structure w passed as input.*

4.7 Merging a Root Copy Vertex with Its Parent Copy

The procedure MergeVertex defined in Figure 4.7 is responsible for merging a root copy vertex w' with its parent copy w. The additional parameter w_{in} helps indicate how the adjacency lists should be joined. The root copy vertex is eliminated by this procedure. The procedure MergeVertex is called from the MergeBicomps procedure defined in Chapter 6 and from the procedure JoinBicomps in Section 4.8.

The correctness of procedure MergeVertex, which is based on Precondition 4.17, is stated in Lemma 4.18. The running time for procedure MergeVertex is stated in Lemma 4.19.

Precondition 4.17 *The procedure MergeVertex is invoked on an embedding $\tilde{G}$ with the following characteristics:*

1. *The vertex structure w is a parent copy, and the vertex structure w' is a root copy of w.*

Figure 4.7: The Procedure MergeVertex

Procedure: MergeVertex
this: Embedding Structure $\tilde{G}$
in: A vertex w
in: An indicator w_{in} (0 or 1) of the link in w that refers to
the edge record used to enter w
in: A root copy vertex w'
out: Embedding Structure $\tilde{G}$ modified by joining the adjacency list of w'
into that of w such that the link[w_{in}] edge record in w' becomes the
new link[w_{in}] edge record in w.

(1) for each edge record e in the adjacency list of w'
(2) Assign w to the neighbor member of the edge record indicated by the twinLink of e

(3) Let $e_{w_{in}}$ be equal to link[w_{in}] of w (result is an edge record unless w is degree 0)
(4) Let $e_{w'_{out}}$ be the link[1 xor w_{in}] edge record of w'
(5) Perform a circular list union of the adjacency lists w' into the
adjacency list of w such that the edge records $e_{w_{in}}$ and $e_{w'_{out}}$
are consecutive and such that the link[w_{in}] edge record in w' becomes the
new link[w_{in}] edge record in w

(6) Assign an indicator of w' to the link[0] and link[1] members of w'
(7) Assign *nil* to the parentLink of the vertex structure w'

2. *The adjacency lists of w, w' and the neighbors of w' conform to Definition 3.2.*

3. *The edge records in the adjacency lists of w and w' are each part of distinct edges that conform with Definition 3.3 or 3.4.*

Lemma 4.18 *As the result of an invocation that complies with Precondition 4.17, the procedure MergeVertex has the following effects:*

1. *In the edges incident to w', the edge records containing w' in the neighbor member are changed to contain w as the neighbor, preserving conformance to Definition 3.2 in the adjacency lists of neighbors of w' after w' is eliminated.*

2. *The adjacency list of w' is inserted (without changing the edge order) into the adjacency list of w such that link[w_{in}] in w indicates the same edge record as*

link$[w_{in}]$ *in* w'.

3. *The root copy* w' *is eliminated from* $\tilde{G}$.

Lemma 4.19 *The MergeVertex procedure runs in* $O(d)$ *time, where* d *is the degree of the root copy vertex* w' *being merged with its parent copy* w.

4.8 Joining All Biconnected Components in the Embedding

The procedure JoinBicomps defined in Figure 4.8 merges all root copy vertices with their parent copies without changing the orientations of any vertices. If the input graph is planar, then this procedure is called by the procedure PlanarEmbed in Chapter 8 after the latter orients all vertices. This is necessary even if the input graph is biconnected since DFS tree root will be represented by a root copy vertex because there are no back edge to ancestors that would result in a merge with its parent copy. If the graph is not planar, then this procedure is called by the Kuratowski subgraph isolator in Chapter 9.

Figure 4.8: The Procedure JoinBicomps

```
Procedure: JoinBicomps
this: Embedding Structure G̃
out: Embedding Structure G̃ modified by merging all root
     copy vertices with their parent copies without changing
     the orientations of any vertices

(1) for each root copy w' from n to 2n − 1
(2)     if the members of the root copy w' have not been cleared
(3)         w ← the parentLink of w'
(4)         MergeVertex(w, 1, w', 0)
```

The correctness of procedure JoinBicomps is established by Lemma 4.20. The run-time analysis for procedure JoinBicomps appears in Lemma 4.21.

Lemma 4.20 *Given an embedding* $\tilde{G}$ *in the oriented embedding invariant state (Definition 3.9), the JoinBicomps procedure merges the adjacency lists of all root copies of each vertex* w *into the parent copy of* w*, preserving Property 3.4.*

Proof. Root copy vertices are only created in the region n to $2n-1$ of V in $\tilde{G}$ by EmbedTreeEdge, which also correctly sets the parentLink of each root copy according to Lemma 4.2. The parentLink is only returned to the *nil* setting by a prior call to MergeVertex, so procedure JoinBicomps simply calls MergeVertex on root copies that have not already been processed by MergeVertex, which according to Lemma 4.18 performs the required adjacency list merge operations, preserving orientation and eliminating root copies. □

Lemma 4.21 *The JoinBicomps procedure runs in $O(n)$ time.*

Proof. The loop performs n iterations. The loop body consists of constant time operations except for the invocation of MergeVertex, which according to Lemma 4.19 is $O(d)$ where d is the degree of a root copy w'. Since each edge incident to a root copy is representative of a unique edge in the input graph, the total work performed by all MergeVertex invocations is $O(n)$. □

5. Walkup

The procedure Walkup defined in Figure 5.1 is invoked by the procedure PlanarityTest in Chapter 7 to support the Definitions 2.3 and 2.4 appearing in Chapter 2. Given a descendant w of the current vertex v being processed, the Walkup marks w as being directly adjacent to the current vertex v. Then, the Walkup stores information in the embedding structure $\tilde{G}$ about the pertinent biconnected components that must be merged in order to embed a back edge (v, w). Specifically, for each separation ancestor z between w and v in $\tilde{G}$, the Walkup ensures that the root copy z^c, where c is a DFS child of z and equal to or an ancestor of w, appears in the pertinentBicompList of vertex z.

The correctness of procedure Walkup is implied in Lemma 5.2 by virtue of the collective operation of all invocations of Walkup indicated in Precondition 5.1. The run-time analysis for procedure Walkup appears in Lemma 5.3.

Precondition 5.1 *Beginning with an embedding structure $\tilde{G}$ in the vertex invariant state (Definition 3.7) for some step v, the procedure PlanarityTest described in Chapter 7 invokes Walkup once per back edge from v to a descendant of v appearing in the input graph G.*

Lemma 5.2 *As a result of the invocations of Walkup described in Precondition 5.1, the embedding structure $\tilde{G}$ is transformed to the embedding invariant state (Definition 3.8).*

Proof. Consider the first invocation of Walkup in step v. Line 1 marks w to indicate that it is adjacent to the current vertex v, and the remainder of the procedure makes the modifications to pertinentBicompList members as follows. Consider the sequence S of parent copies on the DFS tree path between w and v that are separation ancestors in the embedding structure $\tilde{G}$ at the beginning of step v. Both w and all vertices in S are guaranteed to be on the external faces of the biconnected components containing them because they were externally active in the previous step $v + 1$ (or because $\tilde{G}$ is

Figure 5.1: The Walkup Procedure

```
Procedure: Walkup
this: Embedding Structure G̃
in: A vertex w (a descendant of the current vertex v being processed)
out: Embedding Structure G̃ with the adjacentTo member of w
     set to v and the pertinentBicompList of each separation ancestor
     between w and v modified to contain the root copy ancestor of w

(1) Set the adjacentTo member of w equal to v

(2) (x, x_in) ← (w, 1)
(3) (y, y_in) ← (w, 0)

(4) while x ≠ v
(5)       if the visited member of x or y is equal to v
(6)         break the loop

(7)       Set the visited members of x and y equal to v

(8)       if x is a root copy, z' ← x
(9)       else if y is a root copy, z' ← y
(10)      else z' ← nil

(11)      if z' ≠ nil
(12)        Set z equal to the parentLink of z'
(13)        if z ≠ v
(14)          c ← z' − n
(15)          if the lowpoint of c is less than v
(16)            Append z' to the pertinentBicompList of z
(17)          else Prepend z' to the pertinentBicompList of z
(18)        (x, x_in) ← (z, 1)
(19)        (y, y_in) ← (z, 0)

(20)      else (x, x_in) ← GetNextVertexOnExternalFace(x, x_in)
(21)           (y, y_in) ← GetNextVertexOnExternalFace(y, y_in)
```

in the tree edge state given by Definition 3.6 if $v = n - 1$). Moreover, root copies are always on the external face because edges are always embedded from a root copy to its descendants. Therefore, the Walkup is guaranteed to start from a vertex (w or $z \in S$) on the external face of a biconnected component B, and the Walkup is guaranteed to find the root vertex of B due to the correctness of GetNextVertexOnExternalFace (Lemma 4.13).

If a root copy z' is encountered by x or y, and its visited member is not equal to v, then the parent copy z is obtained using the parentLink (according to Definition 3.1), which is set by the procedure EmbedTreeEdge as established in Lemma 4.2. Then, if z is not v, the root copy z' is stored in the pertinentBicompList of z in Lines 15 to 17. Finally, Lines 18 and 19 set the traversal contexts of x and y so the process will repeat starting with z, unless z is v in which case the Walkup loop terminates.

Applying these facts inductively, the Walkup reaches each separation ancestor in S in increasing ancestral distance from w and stores in the separation ancestor's pertinentBicompList the appropriate root vertex of the child biconnected component whose root edge is on the DFS tree path from w to v.

The first invocation of Walkup does not encounter any vertex structures marked visited (i.e. with visited member equal to v). Subsequent invocations of Walkup operate in the same manner described above except if the loop encounters a vertex structure with a visited member already equal to v. Since $\tilde{G}$ began in the vertex invariant state (Definition 3.7), the visited member setting must have occurred during a prior Walkup invocation resulting from Precondition 5.1. By the strong form of induction, the current Walkup invocation can terminate its operation since the remaining separation ancestors to be recorded by the current Walkup invocation have already been recorded by one or more prior invocations of Walkup resulting from Precondition 5.1. □

Lemma 5.3 *The Walkup procedure runs in $O(1)$ amortized time per vertex of the input graph.*

Proof. Lines 1 to 3 are constant time assignments. Line 4 iterates a simple loop

over lines 5 to 21. All operations in the loop body are constant time assignments, expression evaluations and branching instructions except Lines 16 and 17 and the procedure invocations in Lines 20 and 21. Lines 16 and 17 are implemented in constant time using a doubly linked list according to the definition of pertinentBicompList in Table 3.1. The invocations of procedure GetNextVertexOnExternalFace in Lines 20 and 21 are constant time according to Lemma 4.14. Thus, the loop body is $O(1)$ per iteration.

Within each biconnected component B, the Walkup either finds the root vertex or it halts prematurely due to the visitation test in Lines 4 and 5. In either case, the length of the path used to find either a visited vertex or a root copy vertex is no greater than the length of the path along B's external face that would become part of a proper face in the embedding if the back edge to w were embedded. Note that the cost of traversing the path that does not find a visited or root copy vertex is simply charged to the cost of the path that does find the visited or root copy vertex.

Thus, the total cost of all Walkup processing for all vertices is commensurate with the sum of the degrees of the faces in the embedding structure at the end of any step v where all back edges from v to its descendants are embedded. This argument excludes short-circuit edges, which only serve to reduce the total cost of all Walkup calls by shortening the paths traversed. Moreover, if non-planarity is discovered in step v, then one or more back edges are not embedded. In this case, it is easiest to charge the Walkup costs in step v as a one-time $O(n)$ additional cost. Since the extra cost in step v for non-planar graphs is also $O(n)$, the Walkup cost is constant amortized time per vertex. □

6. Walkdown

This chapter covers the procedures that merge biconnected components and embed back edges within the embedding structure. Section 6.1 discusses a procedure that merges a collection of biconnected components that have been selected by the Walkdown procedure, which is discussed in Section 6.2. At the end of Section 6.2, the reasons are explained why the use of short-circuit edges pose no additional memory requirements on the embedding structure.

6.1 Merging the Biconnected Components on the Stack

The procedure MergeBicomps defined in Figure 6.1 processes the embedding structure stack S in $\tilde{G}$ by merging root copies on the stack with their parent copies. Additional stack information indicates whether each root copy's adjacency list should be inverted prior to the merge operation. This procedure is called by the Walkdown procedure in Section 6.2, which also sets the stack entries used by MergeBicomps.

The Walkdown pushes a pair (w, w_{in}) indicating the parent copy vertex w and the direction w_{in} that the Walkdown used to enter w. The Walkdown then descends from w to a pertinent child biconnected component with root denoted w' and chooses a direction w'_{out} in which to proceed. The Walkdown then pushes the pair (w', w'_{out}).

The Walkdown pushes the pairs (w, w_{in}) and (w', w'_{out}) each time it descends from a parent copy vertex to the root copy appearing in a pertinent child biconnected component. Then, when the Walkdown encounters a vertex for which a back edge must be embedded, the biconnected components indicated on the stack combined with the back edge collectively form a new, larger biconnected component in the embedding structure. The Walkdown first calls MergeBicomps to join the pertinent biconnected components listed on the stack, then it embeds the back edge that biconnects them. Moreover, the new proper face formed by adding the back edge includes the paths along each merged biconnected component that the Walkdown traversed to build the stack. Therefore, the Walkdown requires that w' and w be merged in such a way that

Figure 6.1: Merging Pertinent Biconnected Components

Procedure: MergeBicomps
this: Embedding Structure $\tilde{G}$
out: Embedding Structure $\tilde{G}$ with merged biconnected components
corresponding to stack entries made by the Walkdown procedure.

(1) while the stack S in $\tilde{G}$ is not empty
(2) Pop (w', w'_{out}) from the stack S
(3) Pop (w, w_{in}) from the stack S

(4) if w_{in} is equal to w'_{out}
(5) InvertVertex(w')
(6) $e \leftarrow$ DFS tree edge record in w'
(7) Set the sign of edge record e to -1

(8) Remove w' from the front of the pertinentBicompList of w
(9) Using repInParentList of DFS child $c = w' - n$,
Remove c from the separatedDFSChildList of w
(10) MergeVertex(w, w_{in}, w')

the link[w_{in}] edge record in w and the link[w'_{out}] edge record in w' become consecutive in the adjacency list of w, and the link[1 xor w'_{out}] edge record from w' should be the new link[w_{in}] edge record in w.

The correctness of MergeBicomps is established by Lemma 6.2. The run-time analysis for MergeBicomps appears in Lemma 6.3.

Precondition 6.1 *The procedure MergeBicomps is invoked by procedure Walkdown with an embedding $\tilde{G}$ in the embedding invariant state (Definition 3.8), and a stack containing zero or more pairs of 2-tuples of the form (w', w'_{out}) and (w, w_{in}), where w indicates a parent copy vertex structure, w' indicates a root copy of w, and w'_{out} and w_{in} each have a value of 0 or 1.*

Lemma 6.2 *Given an invocation subject to Precondition 6.1, the procedure MergeBicomps has the following effects for each pair of 2-tuples (w', w'_{out}) and (w, w_{in}) appearing in sequence on the top of stack S:*

1. *the tree edge incident to w' is given the sign -1 if w_{in} and w'_{out} are equal.*

2. *the root copy w' is removed from the pertinentBicompList of w.*

3. *the DFS child c in the root edge (w', c) is removed from the separatedDFSChildList of w.*

4. *the adjacency list of the root copy w' is inserted (without changing the edge order) into the adjacency list of its parent copy w such that the link[1 xor w'_{out}] edge record in w' becomes the new link[w_{in}] edge record in w.*

5. *the root copy w' is eliminated from the embedding $\tilde{G}$.*

Proof. By trivial induction, the loop can process a stack of any size. Consider the behavior of the loop body for each pair (w', w'_{out}) and (w, w_{in}) popped from the stack. Clearly the first requirement is met by Lines 4 and 7. Lines 8 and 9 meet the second and third requirements. Due to the invocation of InvertVertex when w_{in} and w'_{out} are equal, link[w_{in}] in w' is guaranteed to indicate the edge record that was indicated by link[1 xor w'_{out}] in w' when Line 2 was executed. Thus, by Lemma 4.18, the invocation of MergeVertex meets the fourth requirement. Finally, according to Lemma 4.18, MergeVertex eliminates the root copy w', meeting the fifth requirement. □

Lemma 6.3 *The procedure MergeBicomps runs in $O(1)$ amortized time per vertex of the input graph.*

Proof. Line 1 initiates a simple loop over the loop body in Lines 2 to 10, iterating once per root copy vertex w' on the stack. All operations in the loop body are constant time except the invocations of InvertVertex and MergeVertex, which according to Lemmas 4.16 and 4.19 operate in $O(d)$ time where d is the degree of w'. However, MergeBicomps is only invoked to process edges incident to root copy vertices. After MergeVertex, all of the edges incident to the root copy become incident to the parent copy, so they are never processed again by MergeBicomps. Thus, the loop achieves $O(1)$ amortized time due to the edge limit given by Corollary 1.5. □

6.2 The Walkdown Procedure

Given the root v' of a singleton biconnected component B containing the only the tree edge (v', c), the procedure Walkdown defined in Figure 6.2 embeds back edges from the root v' to the descendants of c. These descendants appear in pertinent biconnected components that are merged into B such that all of the embedded back edges become part of B. The Walkdown procedure is called by the procedure PlanarityTest described in Chapter 7.

The correctness of procedure Walkdown is established by a number of lemmas appearing below, culminating in Theorem 6.23. The run-time analysis for the Walkdown appears in Lemma 6.24.

In Figure 6.2, Line 2 initiates an outer loop to handle the fact that traversal around the external face of a biconnected component can proceed from the root copy v' in two directions, indicated by v'_{out}. The inner traversal loop starting at Line 3 begins at v' with the edge indicated by the link$[v'_{out}]$ member of v'. The inner traversal loop proceeds to embed back edges and merge biconnected components as necessary, and it adds short-circuit edges to remove inactive vertices from the external face. The first iteration of the loop with $v'_{out} = 0$ proceeds until it returns to v' or until it reaches a stopping vertex. In the latter case, a second iteration of the loop with $v'_{out} = 1$ operates in the same manner, terminating when a stopping vertex is encountered. For each loop iteration, the Walkdown maintains the embedding invariant state (Definition 3.8).

Lemma 6.4 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown traversal loop maintains as an invariant condition Property 3.1, which states that $\tilde{G}$ is a collection of planar biconnected components.*

Proof. The Walkdown only changes the biconnected components in $\tilde{G}$ to embed a back edge (Lines 6 and 7) or a short-circuit edge (Line 27). A short-circuit edge is always embedded between two vertices in the same biconnected component, so it does not affect Property 3.1. At the beginning of an iteration that embeds a back edge (v', w), the stack S contains the root copies of all pertinent child biconnected

Figure 6.2: The Walkdown Procedure

Procedure: Walkdown
this: Embedding Structure $\tilde{G}$
in: A root copy v' of the current vertex (with DFS child c)
out: Embedding Structure $\tilde{G}$ containing back edges to descendants of v',
merged biconnected components and short-circuit edges

(1) Clear the stack S in $\tilde{G}$
(2) for v'_{out} in $\{0, 1\}$
(3) $(w, w_{in}) \leftarrow$ GetNextVertexOnExternalFace(v', 1 xor v'_{out})
(4) while $w \neq v'$
(5) if the adjacentTo member of w is equal to v
(6) MergeBicomps()
(7) EmbedBackEdge(v', v'_{out}, w, w_{in})
(8) Set the adjacentTo member of w equal to n

(9) if |pertinentBicompList of w| > 0
(10) Push (w, w_{in}) onto the stack S
(11) $w' \leftarrow$ value of the first element of pertinentBicompList of w
(12) $(x, x_{in}) \leftarrow$ GetNextVertexOnExternalFace(w', 1)
(13) $(y, y_{in}) \leftarrow$ GetNextVertexOnExternalFace(w', 0)

(14) if VertexActiveStatus(x) returns 'Internally Active'
(15) $(w, w_{in}) \leftarrow (x, x_{in})$
(16) else if VertexActiveStatus(y) returns 'Internally active'
(17) $(w, w_{in}) \leftarrow (y, y_{in})$
(18) else if adjacentTo of x equals v or |pertinentBicompList of x| > 0
(19) $(w, w_{in}) \leftarrow (x, x_{in})$
(20) else $(w, w_{in}) \leftarrow (y, y_{in})$

(21) if w equals x, $w'_{out} \leftarrow 0$
(22) else $w'_{out} \leftarrow 1$
(23) Push (w', w'_{out}) onto the stack S

(24) else if VertexActiveStatus(w) returns 'Inactive'
(25) $(w, w_{in}) \leftarrow$ GetNextVertexOnExternalFace(w, w_{in})
(26) if the stack S is empty and the lowpoint of $c = v' - n$ is less than v
and the adjacentTo member of w is not equal to v
(27) EmbedShortCircuitEdge(v', v'_{out}, w, w_{in})

(28) else break the 'while' loop

(29) if the stack S is non-empty or w equals v', break the 'for' loop

components to which the Walkdown descended in traversing external face paths from v' to w. These root copies, along with other information identifying the specific paths traversed, were pushed onto S (Lines 9 and 23) during prior iterations. By Lemma 6.2, the procedure MergeBicomps joins the root copies on the stack S with their corresponding parent copies. The resulting connected component contains the concatenation of the external face paths traversed, which form a cycle when combined with the new back edge (v', w) form a cycle. Thus, once the new back edge is embedded (Line 7), it appears in a single biconnected component. Therefore, the Walkdown does not change $\tilde{G}$ in a way that affects Property 3.1. □

Lemma 6.5 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown traversal loop maintains as an invariant condition Property 3.2, which states that all internally and externally active vertices appear on the external faces of biconnected components in $\tilde{G}$.*

Proof. A short-circuit edge can remove a vertex from the external face, but only if it is inactive according to the test in Line 24. Moreover, both endpoints of a short-circuit edge remain on the external face of a single biconnected component. As well, both endpoints of a back edge and every vertex into which a root copy is merged remain on the external face. Thus, since there is an edge directly connecting each root copy r' with the merge point or back edge endpoint in the same biconnected component as r', and since the new back edge is embedded such that it forms a proper face with the aforementioned edges, the merging of biconnected components and the embedding of back edges do not remove any vertices from the external face. □

Lemma 6.6 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown traversal loop maintains as an invariant condition Property 3.3, which states that external face edges are indicated by the link[0] and link[1] members of external face vertices.*

Proof. Follows from Lemmas 4.5 and 4.8 and the proof of Lemma 6.5. □

Lemma 6.7 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the inner Walkdown traversal loop maintains as an invariant condition Property 3.4, which states how to obtain the clockwise orientation of all vertex adjacency lists.*

Proof. When a back edge $(v, w) \in E(G)$ is embedded in $\tilde{G}$, its representative edge records are placed in the adjacency lists of a root copy v' and the parent copy w such that the new edge appears on the external face in place of the path traversed by the Walkdown to travel from v' to w. A short-circuit edge embedding performs the same adjacency list modifications. Thus, embedding an edge does not affect vertex orientation of its endpoints.

Other than v', root copies are only affected by being destroyed. A root copy v' is only affected by edge addition. Thus, clockwise vertex orientation is preserved for root copies.

For parent copies, the only other Walkdown operation that modifies the adjacency lists is the merging of biconnected components. According to Lemma 6.2, the adjacency list of r after being merged with a root copy r' is correctly formed and has the same orientation. Therefore, we can consider whether Property 3.4 is preserved by the procedure MergeBicomps. The property is true by assumption prior to an invocation of MergeBicomps. Inductively, assume the embedding has Property 3.4 and the embedding contains a connected component R whose vertex of least DFI is represented by a root copy r'. Let r denote the parent copy of r' and let q' denote the root of the biconnected component containing r. Let r_{in} be 0 or 1 to indicate the direction of the Walkdown's entry into r. Let r'_{out} be 0 or 1 to indicate the direction of the Walkdown's exit from r'.

Case 1: Let the parity of r be 0 such that r and q' have the same orientation, and let r_{in} differ from r_{out}. Since r_{in} differs from r_{out}, the orientations of r and r' are equal. Thus, r' and q' have the same orientation, so Property 3.4 is preserved if the orientations of descendants of r' in R are not changed. Since the parity of r is 0 and the sign of the child tree edge incident to r' is not changed from the initial value of 1, Property 3.4 is preserved.

Case 2: Let the parity of r be 0 such that r and q' have the same orientation, and let r_{in} be equal to r_{out}. Since r_{in} equals r_{out}, the orientations of r and r' are opposed. Thus, r' and q' have opposite orientations, so Property 3.4 is preserved if the orientations of descendants of r' in R are changed. Since the parity of r is 0 and the sign of the child tree edge incident to r' is changed to -1 after r' is inverted, Property 3.4 is preserved.

Case 3: Let the parity of r be 1 such that r and q' have opposite orientations, and let r_{in} differ from r_{out}. Since r_{in} differs from r_{out}, the orientations of r and r' are equal. Thus, r' and q' have opposite orientations, so Property 3.4 is preserved if the orientations of descendants of r' in R are changed. Since the parity of r is 1 and the sign of the child tree edge incident to r' is not changed from the initial value of 1, Property 3.4 is preserved.

Case 4: Let the parity of r be 1 such that r and q' have opposite orientations, and let r_{in} equal r_{out}. Since r_{in} equals r_{out}, the orientations of r and r' are opposed. Thus, r' and q' have the same orientation, so Property 3.4 is preserved if the orientations of descendants of r' in R are not changed. Since the parity of r is 1 and the sign of the child tree edge incident to r' is changed to -1 after r' is inverted, Property 3.4 is preserved. □

Lemma 6.8 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown traversal loop maintains as an invariant condition Property 3.5, which states that all unnecessary root copies are eliminated.*

Proof. A root copy p^c is only unnecessary if a back edge has been embedded between an ancestor of p and a vertex in the DFS subtree rooted by c. According to the proof of Lemma 6.4, the embedding of such a back edge is immediately preceded by the merging of p^c with p in the procedure MergeBicomps. According to Lemma 6.2, p^c is eliminated. □

Lemma 6.9 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown traversal loop maintains as an invariant condition*

Property 3.6, which states that external activity of a vertex p is dynamically updated by removing the child c from the separatedDFSChildList of parent p when the root copy p^c is eliminated.

Proof. According to Definition 2.1, the external activity of a vertex p is determined by its leastAncestor, which does not change, and the external activity of biconnected components containing root copies of p. Lemma 4.10 proves that the external activity of biconnected component with root edge (p^c, c) is determined from the lowpoint of c. According to Lemma 4.11, the procedure VertexActiveStatus uses the lowpoint of the first element of the separatedDFSChildList because it is sorted by the children's lowpoint values, and a child c is removed from the separatedDFSChildList of parent p when p^c is eliminated by Lemma 6.2. □

Lemma 6.10 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown traversal loop maintains as an invariant condition Property 3.7, which states that the embedding maintains a count of the number of embedded edges.*

Proof. According to Lemmas 4.5 and 4.8, embedding a new edge implies an increment of the value of m in $\tilde{G}$ due to the definition of edge representation (Definition 3.3). □

Lemma 6.11 *Starting with an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown traversal loop maintains as an invariant condition Property 3.8, which states that the pertinence of vertices is dynamically updated in a vertex w and its separation ancestors when the back edge (v, w) is embedded.*

Proof. According to Definition 2.4, a vertex w is pertinent if there is an unembedded back edge (v, w) or if there is a pertinent biconnected component with root w'. By Definition 3.8, the adjacentTo member of w is equal to v if there is an unembedded back edge (v, w), and the pertinentBicompList of w contains a root copy of w for

each pertinent child biconnected component. Immediately after embedding a back edge, the Walkdown traversal loop assigns n to the adjacentTo member of w (Line 8). According to Lemma 6.2, the procedure MergeBicomps removes a root copy w' from the pertinentBicompList of its parent copy w when w' is eliminated. □

Lemma 6.12 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8), the Walkdown procedure maintains the embedding invariant state of $\tilde{G}$.*

Proof. Follows directly from the proofs of Lemmas 6.4, 6.6, 6.5, 6.7, 6.8, 6.9, 6.10 and 6.11. □

For the Walkdown invocation on a root edge (v', c), the traversal loop invariants given above are clearly useful when the Walkdown is able to embed all back edges from v to descendants of c. Thus, we focus on Conditions 1 and 2, which describe the conditions under which the Walkdown fails to embed one or more back edges from v to descendants of c.

Condition 1. *Given an embedding structure $\tilde{G}$ and a root copy v' in $\tilde{G}$ with DFS child c, the Walkdown returns with a non-empty stack (which implies that one or more back edges from v to descendants of c have not been embedded).*

Condition 2. *Given an embedding structure $\tilde{G}$ and a root copy v' in $\tilde{G}$ with DFS child c, the Walkdown returns with an empty stack prior to embedding all back edges from v to descendants of c.*

If Condition 1 occurs, then the input graph G contains as a minor the graph appearing in Figure 6.3(a). If Condition 2 occurs, then the input graph G contains as a minor the graph in Figure 6.3(b). It is easy to prove that the graph minor in Figure 6.3(a) indicates the non-planarity of the input graph, as shown in Lemma 6.13.

Lemma 6.13 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown invocation terminates with a non-empty stack (Condition 1), then the input graph G is not planar.*

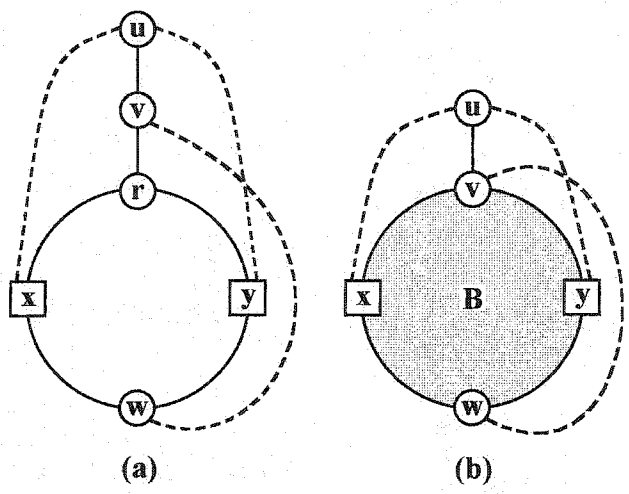


Figure 6.3: Graph Minors for Non-planarity Conditions of the New Planarity Test. (a) Non-planarity Minor A for Condition 1. (b) Non-planarity Minor for Condition 2.

Proof. Edge contraction and deletion can be used on the embedding structure $\tilde{G}$, along with the addition of certain unembedded edges represented in $\tilde{G}$ as vertex activity, to produce the minor appearing in Figure 6.3(a). The vertex v is the current vertex, and u represents all unprocessed ancestors of v. The edge (u, v) is representative of the DFS tree path from v to ancestors of interest in u. The vertices x and y are the externally active, and the dashed edges (u, x) and (u, y) are representative of unembedded back edges from the input graph that are indicated in $\tilde{G}$ by the external activity of x and y. Moreover, the specific edges to ancestors of v may be incident to descendants of x and y that have been edge contracted into x and y in the diagrams shown. Thus, x and y may be representative of subgraphs of the input graph. Similarly, vertex w is or has a descendant adjacent to the current vertex v by a back edge (v, w), which is also depicted with a dashed line because the Walkdown procedure failed to embed it.

In Figure 6.3(a), there is a biconnected component of central interest rooted by a root copy of r in $\tilde{G}$. It has an external face consisting of the cycle $(r, \ldots, x, \ldots, w, \ldots, y, \ldots, r)$. For the purpose of simplicity, root copy vertices are not explicitly depicted as they are simply part of the representation of the parent copy vertex while it is a cut vertex in the partial embedding $\tilde{G}$. The vertices within the biconnected

component with root r can simply be ignored, and vertices along the external face other than r, x, w, and y can be edge contracted into x and y (such that the cycle $(r, \ldots, x, \ldots, w, \ldots, y, \ldots, r)$ is maintained). The edge (v, r) is representative of the depth first search tree path between v and r.

The non-empty stack indicates that the Walkdown traversal loop has descended to a pertinent child biconnected component, but the loop has encountered a stopping vertex prior to finding a pertinent vertex w. According to Lines 14-20, selecting a path to a stopping vertex can only occur if both external face paths lead from the child biconnected component root to stopping vertices x and y. Thus, both x and y in Figure 6.3(a) must be stopping vertices. Since the input graph G contains as a minor the graph in Figure 6.3(a), which in turn is a $K_{3,3}$, the input graph is not planar. □

As mentioned previously, Figure 6.3b is a graph minor representing the state of the embedding structure $\tilde{G}$ when Condition 2 occurs. Its interpretation is similar to the description given in Lemma 6.13. The biconnected component B on which the Walkdown was called has an external face represented by the cycle $(v, \ldots, x, \ldots, w, \ldots, y, \ldots, v)$. Vertices x and y represent the two stopping vertices that halted the Walkdown traversal loop in each direction. Vertex u represents all unprocessed ancestors of v, and the edge (u, v) represents a depth first search tree path. The dashed edges represent unembedded back edges from the input graph that are indicated in the embedding structure $\tilde{G}$ by internal or external activity.

The non-planarity of G is more difficult to establish when the Walkdown fails to embed a back edge due to Condition 2. In Figure 6.3(b), it is clear that the algorithm would surround an externally active vertex if it were to proceed to w and embed the edge (v, w). However, the figure does not contain a sufficient portion of the input graph to establish non-planarity. In particular, it is not clear why the dashed edge (v, w) cannot simply be embedded inside the biconnected component. The lemmas that follow will prove that the Walkdown does indeed embed (v, w) inside of the biconnected component B before encountering both stopping vertices, except when

sufficient additional structure exists to establish the non-planarity of the input graph.

A w *candidate* is a pertinent vertex on the lower path between x and y along the bounding cycle of the biconnected component with root edge (v', c) (see, for example, the biconnected component denoted B in Figure 6.3(b)). Lemma 6.14 proves that if the Walkdown fails to embed a back edge due to Condition 2, the stopping vertices x and y must be distinct, and Lemma 6.15 proves that if the Walkdown fails to embed a back edge to a w candidate (or one of its descendants), then the w candidate must be on the external face of B.

Lemma 6.14 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2, which states that it fails to embed a back edge between v' and a descendant of c, then the stopping vertices x and y are distinct.*

Proof. By contradiction, suppose x and y are the same vertex w. If w is adjacent to v, then the Walkdown embeds the back edge to w before halting because Lines 5-8 run before Line 28. Furthermore, a stopping vertex has no pertinent child biconnected components. Hence, there are no DFS descendants of w that are adjacent to v, so the stopping vertex does not prevent the Walkdown from visiting any vertices adjacent to v, which contradicts the existence of a w candidate. □

Lemma 6.15 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2, which states that it fails to embed a back edge between v' and a w candidate (or one of its descendants), then the w candidate must be on the external face of B.*

Proof. In the first step of the main algorithm (see Figure 2.1), only tree edges are embedded, so all vertices are on the external faces of the biconnected components containing them. In subsequent steps, all pertinent vertices are on the external face because they were externally active in the prior step. Moreover, a vertex w can only

be placed inside a biconnected component by embedding a short-circuit edge between its neighbors along the external face, which in turn would imply that w is inactive. However, a vertex w cannot be inactive if it is pertinent. □

To simplify future results, Lemma 6.16 and Lemma 6.17 show that pertinent child biconnected components of a w candidate can be edge contracted into w without affecting conclusions about the correctness of the algorithm.

Lemma 6.16 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2, then we can consider the behavior the Walkdown as if all internally active descendant biconnected components of each w candidate were edge contracted into the w candidate.*

Proof. Once the algorithm merges an internally active child biconnected component B' into B at the cut vertex w, it traverses all pertinent descendant biconnected components of B' before w is encountered again. Since B' is not externally active, no stopping vertex will be encountered in this process. Furthermore, all vertices directly adjacent to v must be on the external faces of B' and its descendant biconnected components (as stated in Lemma 6.15). Thus, the back edges from v to vertices in B' and its descendant biconnected components are embedded consecutively in the cyclic order of v. Since B' and its descendant biconnected components were planar subgraphs prior to this operation, and the new back edges are embedded without edge crossings, edge contracting B' and its descendant biconnected components into the w candidate can be done without affecting the planarity of $\tilde{G}$ and results in a single edge (v, w). □

Lemma 6.17 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2 and a w candidate has a pertinent externally active child biconnected component, then the input graph G is not planar.*

Proof. The graph minor appearing in Figure 6.4 is the same as the Walkdown halting minor in Figure 6.3(b) except that z and its incident edges have not been contracted into w. The edge (w, z) is the root edge of the pertinent externally active child biconnected component, and all DFS descendants of z have been edge contracted into z. Since the input graph contains as a minor the graph in Figure 6.4, which contains a $K_{3,3}$, the input graph is not planar. □

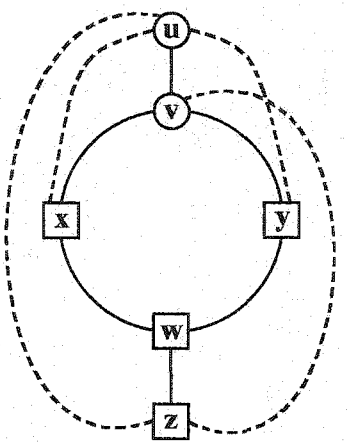


Figure 6.4: Non-planarity Minor B for Lemma 6.17

Figure 6.4 is the first graph minor extension of Figure 6.3(b) with enough additional structure to establish the non-planarity of the input graph. In this case, one can think of the unembedded back edge (v, w) in Figure 6.3(b) as representing a subgraph that includes z, so embedding (v, w) inside of the biconnected component B could not be accomplished without placing an externally active vertex inside the bounding cycle of B. All future cases exploit a second type of additional structure to establish the non-planarity of Figure 6.3(b). The additional structure is a path through B that would result in an edge crossing if (v, w) were also embedded inside B.

An *x-y path* is a path embedded through the biconnected component B depicted in Figure 6.3(b) that excludes the root of B and a specific w candidate, and which has one endpoint p_x along the external face path $(v \ldots x \ldots w)$ and the other endpoint p_y along the external face path $(v, \ldots, y, \ldots, w)$. A *high x-y path* is an x-y path in which p_x is on the external face path between and excluding the root of B and x or

p_y is on the external face path between and excluding the root of B and y (or both). A *low x-y path* is an x-y path that is not a high x-y path. The *highest x-y path* is an x-y path such that, for every vertex z along the x-y path including the points of attachment p_x and p_y, an edge crossing does not result from the addition of an edge inside the biconnected component from z to the root copy of v. The first condition under which a blocking x-y path results in the conclusion that the input graph is non-planar appears in Lemma 6.18.

Lemma 6.18 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2 and the biconnected component B rooted by v' has a high x-y path, then the input graph G is not planar.*

Proof. Suppose p_x is a high point of attachment for the x-y path. The graph minor appearing in Figure 6.5 is the same as the Walkdown halting minor in Figure 6.3(b) except for the addition of z and the edge between z and y. Vertex z, which represents the high point of attachment p_x, has not been edge contracted into x. The point of attachment p_y has been edge contracted into y along with all other external face vertices along the path $(v, \ldots, y, \ldots, w)$, excluding v and w. External face vertices along the path $(z, \ldots, x, \ldots, w)$ are edge contracted into x, excluding z and w. Any external face vertices (excluding v) along the path represented by edge (v, z) are edge contracted into z, as are all vertices embedded inside of B. Since the input graph contains as a minor the graph in Figure 6.5, which contains a $K_{3,3}$, the input graph is not planar. The case in which p_y is the high point of attachment is symmetric. □

Lemmas 6.19 and 6.20 establish the two remaining non-planarity minors. Both involve a low x-y path that prevents the embedding of a back edge between v and a w candidate inside the biconnected component B.

Lemma 6.19 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2 and the biconnected component B rooted by v' has an x-y path*

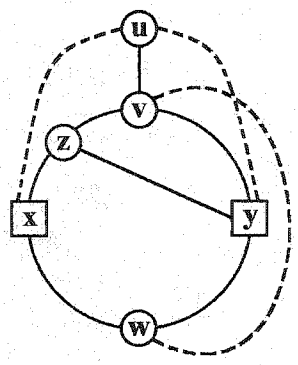


Figure 6.5: Non-planarity Minor C for Lemma 6.18

that contains a single endpoint z of a second path p of the form $(v', \ldots, z)$, where all vertices in the path (except v') are embedded inside B, then the input graph G is non-planar.

Proof. The graph minor appearing in Figure 6.6 is the same as the Walkdown halting minor in Figure 6.3(b) except that the x-y path and the second path p are depicted. All vertices inside B are edge contracted into z. Since the input graph contains as a minor the graph in Figure 6.6, which contains a $K_{3,3}$, the input graph is not planar. □

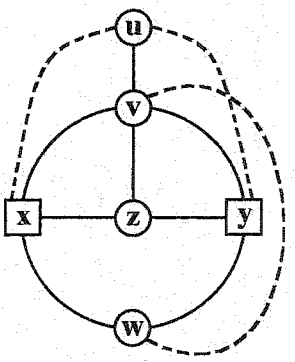


Figure 6.6: Non-planarity Minor D for Lemma 6.19

Lemma 6.20 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2 and the biconnected component B rooted by v' has an x-y path such*

that the lower external face path (p_x, ..., w, ..., p_y) includes an externally active vertex z (distinct from p_x and p_y), then the input graph G is non-planar.

Proof. The graph minor appearing in Figure 6.7 is the same as the Walkdown halting minor in Figure 6.3(b) except that an edge (u, w) has been added and the x-y path is depicted. The externally active vertex z either is or has been edge contracted into the w candidate, but the dashed edge (u, w) indicates the external activity of z. Since the input graph contains as a minor the graph in Figure 6.7, which is a K_5, the input graph is not planar. □

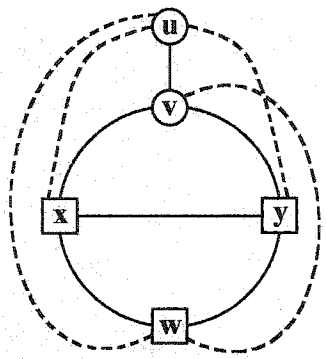


Figure 6.7: Non-planarity Minor E for Lemma 6.20

Now that all of the non-planarity minors have been presented, the next two lemmas show some conditions which lead to a contradiction in the assumption that the Walkdown failed due to Condition 2. Lemma 6.21 hypothesizes, for the purpose of contradiction, the existence of an x-y path that does not meet the criteria set forth in Lemmas 6.18, 6.19 and 6.20. Lemma 6.22 hypothesizes, for the purpose of contradiction, the existence of a w candidate for which there is no blocking x-y path and which does not meet the criterion of Lemma 6.17.

Lemma 6.21 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2 and the biconnected component B rooted by v' has an x-y path, then the highest x-y path must either be a high x-y path, or it must contain an internal*

vertex z that is an endpoint of a second path of internal vertices leading to the root v' of B, or the external face path $(p_x, \ldots, w, \ldots, p_y)$ must include an externally active vertex z (distinct from p_x and p_y).

Proof. By contradiction, suppose none of the three conditions holds for the highest x-y path P. Since by assumption there is no path from the root of B to an internal vertex z along P, there is no DFS tree path from the root of B to the internal vertices of P. Hence, prior to the Walkdown invocation on v', the cycle $(p_x, \ldots, w, \ldots, p_y, \ldots, p_x)$ form the bounding cycle of a biconnected component B' rooted at either p_x or p_y. Because P is a low x-y path by assumption, B' cannot contain the stopping vertices x and y except at the points p_x and p_y, respectively. Let a be the minimum of p_x and p_y, and let b be the maximum of p_x and p_y such that a is a DFS ancestor of b, and b is externally active because it either is a stopping vertex or it has a stopping vertex as a descendant. During step v, the Walkdown merges the biconnected component rooted by a. In Lines 14-20, the Walkdown chooses to proceed with an internally active vertex if one is available. Since b is externally active, but by assumption there are no externally active vertices on the external face path $(a, \ldots, w, \ldots, b)$ (excluding a and b), the Walkdown would proceed along the path to w. Thus, we contradict the existence of the x-y path since the Walkdown would flip the biconnected component if necessary to embed (v, w) first such that vertex w and the edge (v, w) would be embedded inside the biconnected component B when the connection is later made between v' and b or one of its descendants. □

Lemma 6.22 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, if the Walkdown terminates under Condition 2 and the biconnected component B rooted by v' has a w candidate, then B must have an x-y path relative to w or w must have an externally active pertinent child biconnected component.*

Proof. By contradiction, suppose there exists a w candidate for which neither condition holds. Since by assumption there is no x-y path, there is no x-y path consisting of

DFS tree edges. Thus, since w must also be on the external face by Lemma 6.15, w is both a DFS ancestor of x or y (possibly both) and a cut vertex in $\tilde{G}$ at the beginning of step v. Therefore, w must be visited by Walkdown in order to attach the pertinent child biconnected component(s) leading to x or y (or both). The Walkdown embeds any required back edge and attaches all internally active child biconnected components prior to processing externally active child biconnected components (x and y are externally active). The visitation of w and all of its internally active descendants contradicts the assumption that w is a w candidate since by assumption w has no externally active pertinent child biconnected components at the end of step v. □

Theorem 6.23 *Given an embedding structure $\tilde{G}$ in the embedding invariant state (Definition 3.8) and a root copy v' in $\tilde{G}$ with DFS child c, the actions of the Walkdown result in an embedding structure $\tilde{G}$ in the embedding invariant state. If the graph G is planar or if neither Condition 1 or 2 occur, then $\tilde{G}$ contains all back edges from the input graph G between v and descendants of c.*

Proof. If Walkdown fails to embed a back edge, it must be due to Condition 1 or Condition 2 since the only other loop termination occurs on Line 4 if traversal returns to v', which cannot occur if any vertex on the external face is pertinent. If the failure is due to Condition 1, then Lemma 6.13 proves that the input graph is not planar. Hence, consider the case when the algorithm returns NONPLANAR due to a Walkdown failure on Condition 2.

A w candidate, which has an unembedded back edge to v at the end of step v, either has or does not have a blocking x-y path. If there is no blocking x-y path, then Lemma 6.22 proves that w must have an externally active child biconnected component, which proves that the input graph is non-planar by Lemma 6.17. If there is a blocking x-y path, then it is either high or low. If it is a high x-y path, the Lemma 6.18 proves that the input graph is non-planar. If it is a low x-y path, then Lemma 6.21 proves that one of Lemmas 6.19 or 6.20 must hold, which in turn proves that the input graph is non-planar.

Thus, if the Walkdown fails to embed one or more back edges, then G is non-planar. The theorem follows as contrapositive since Lemma 6.12 proves that $\tilde{G}$ is a representation of the combinatorial planar embedding of a subgraph of G consisting of all embedded edges. □

Finally, we have the run-time analysis for the Walkdown appearing in Lemma 6.24. This is followed by Lemma 6.25, which proves that the use of short-circuit edges add no additional memory requirements beyond the edge bound established in Corollary 1.5.

Lemma 6.24 *The procedure Walkdown runs in $O(1)$ amortized time per vertex of the input graph.*

Proof. The Walkdown procedure is invoked once per depth first search tree edge. Line 1 occurs in constant time. Line 2 performs two iterations of remainder of the Walkdown pseudo-code. Line 3 involves a simple assignment and an invocation of GetNextVertexOnExternalFace, which is $O(1)$ by Lemma 4.14. Line 29 is a constant-time comparison and branch operation. Thus, the total cost of all executions of Lines 1-3 and 29 is $O(n)$.

Line 4 initiates a simple loop over the loop body in Lines 5 to 28. The loop body contains invocations of MergeBicomps, EmbedBackEdge, GetNextVertexOnExternalFace, VertexActiveStatus, and EmbedShortCircuitEdge, all of which achieve $O(1)$ time by Lemmas 6.3, 4.6, 4.14, 4.12, and 4.9, respectively. The loop body also invokes a standard $O(1)$ stack push operation on Lines 10 and 23. Line 11 is $O(1)$ because each pertinentBicompList is a linked list. Since the remainder of the operations in the loop body are simple constant time operations, the loop body achieves $O(1)$ time per iteration.

For the inner traversal loop starting at Line 4, the iterations can be partitioned into a set of iterations up to and including the embedding of the first back edge, sets of iterations up to and including the embedding of each subsequent back edge, and a final set of iterations after the last back edge is embedded. For each set of iterations prior to and including the embedding of a back edge, the Walkdown iterates through

vertices along a path that, together with the back edge, will form a new proper face in the embedding structure. Therefore, except for the final set of iterations, the total cost of all loop iterations is commensurate with the sum of the degrees of the faces in the embedding, which is $O(n)$ in total. If the final set of iterations is non-empty, then it consists of zero or more iterations, each of which short-circuits an inactive vertex, followed by a final iteration that discovers either a stopping vertex or the root vertex v'. The total cost of all short-circuiting iterations is $O(n)$ because each vertex is only short-circuited once, after which it no longer appears on the external face. The total cost of the final iteration in each Walkdown is $O(n)$.

Thus, the cost of all Walkdown operations is $O(n)$ in total. This cost can be amortized over all vertices, resulting in $O(1)$ amortized time performance per vertex. □

Lemma 6.25 *Short-circuit edges are embedded within the edge limit specified by Corollary 1.5.*

Proof. The number of short-circuit edges embedded never exceeds n because each is associated with the inactive vertex it removes from the external face. Moreover, the inactive vertex is the neighbor of the endpoints of the short-circuit edge, which forms a degree three proper face when it is added to the embedding. Since an edge borders on two faces, we consider whether the opposing face (currently the external face) can be degree two. First, suppose w is pertinent. A short-circuit edge (v', w) is not added if there is an unembedded back edge (v, w) because the back edge will be added as the first action of the next loop iteration (see Line 5 and the last conditional test in Line 26). If w has a pertinent child biconnected component that is merged in a later iteration of the same traversal loop, then the new proper face bordering the short-circuit edge is at least degree three (the new edge (v', d) to a descendant of w, the short-circuit edge (v', w) and the external face path(s) from w to d). Now, suppose w is inactive. Traversal continues to the external face successor s of the descendant endpoint w, which cannot be equal to v' because the Walkdown only adds short-circuit edges to a biconnected component B if it is externally active. Thus, the Walkdown

either embeds an edge (v', s) or it merges a pertinent child biconnected component onto s. In either case, the short-circuit edge would become part of a new proper face with degree three or higher. Therefore, a short-circuit edge can only border a degree two face if w is a stopping vertex, in which case the short-circuit edge remains along the external face of B. In addition, the external face of the biconnected component B can be degree two only if both Walkdown traversals are stopped by the same vertex w because the external face of B must be degree three or higher if the stopping vertices for B are distinct. Moreover, the external face of B is degree three or higher prior to the addition of the final edge to B in the Walkdown invocation, which must be the short-circuit edge (v', w) if the external face of B becomes degree two as a result of its addition. Thus, prior to the embedding of (v', w), the biconnected component B conformed to Corollary 1.5, and we need only account for the space used by (v', w). Although the input graph is restricted to $3n-5$ edges and our data structure contains space for at least $3n$ edges, our argument does not rely on the space for the extra five edges. Instead, we account for the edge records required by (v', w) using the space for the unembedded back edge e between an ancestor of v and a vertex in the DFS subtree rooted by w. Such a back edge e must exist since w is externally active. If the back edge e is embedded in a later step of the algorithm, then the degree two external face of B is divided by the paths from v to the ancestor and from w to the ancestor such that both edges formerly along the degree two external face are now bordered by faces of degree three or higher. On the other hand, if a non-planarity condition is discovered prior to the embedding of back edge e, then our algorithm deletes the short-circuit edges prior to attempting the isolation of a Kuratowski subgraph. □

7. New Planarity Testing Algorithm

The procedure PlanarityTest defined in Figure 7.1 is the functional entry point of the planarity testing algorithm. This procedure can be called independently, but is also called by procedure PlanarEmbed discussed in Chapter 8 and procedure IsolateKuratowskiSubgraph discussed in Chapter 9.

Figure 7.1: The Edge Addition Planarity Testing Algorithm

Procedure: PlanarityTest
in: Graph G
out: An indication of whether G is planar or non-planar and the embedding structure $\tilde{G}$ (in embedding invariant state) at the time when the planarity of G is determined.

(1) $\tilde{G} \leftarrow$ Initialize(G)
(2) for each vertex $v \in G$,
(3) for each DFS child c of v in G,
(4) $\tilde{G}$.EmbedTreeEdge(v, c)

(5) for each vertex v from $n(G) - 1$ to 0 in descending order
(6) Set v in $\tilde{G}$ equal to the loop variable v

(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) if |pertinentBicompList of $\tilde{G}.V[c]| > 0$
(11) $v' \leftarrow c + \tilde{G}.n$
(12) $\tilde{G}$.Walkdown(v')

(13) for each back edge of G incident to v and a descendant w
(14) if the adjacentTo member of w in $\tilde{G}$ equals v
(15) return (NONPLANAR, $\tilde{G}$)

(16) return (PLANAR, $\tilde{G}$)

Theorem 7.1 establishes the correctness of the PlanarityTest procedure. It is

based on numerous lemmas also appearing in Chapter 6. Theorem 7.2 gives the running time of the PlanarityTest procedure.

Theorem 7.1 *On an input graph G with at most kn edges for some constant $k \geq 3$, the procedure PlanarityTest returns PLANAR if and only if the input graph G is planar, and it returns an embedding structure $\tilde{G}$ in embedding invariant state (Definition 3.8) containing all edges of G unless there exists a biconnected component in $\tilde{G}$ on which a Condition 1 or 2 failure occurred.*

Proof. The tree edges are embedded by the first loop, leaving only the back edges. If, for each vertex v, the back edges from v to its descendants are embedded, then all back edges are embedded since a back edge from a vertex to an ancestor u is embedded in step u. The loop in Lines 9 to 12 iterates each DFS child c of v, invoking Walkdown to embed back edges from v to descendants of c. If G is planar, then Theorem 6.23 proves that these invocations embed the back edges from v to its descendants at every step v. Thus, Line 16 returns PLANAR if G is planar.

Conversely, if PlanarityTest returns PLANAR, then G is indeed planar. The loop in Lines 13 to 15 ensures that, for each vertex v, all back edges from v to its descendants have been embedded, and the result in $\tilde{G}$ at each step is, according to Lemma 6.12, in the embedding invariant state. To return PLANAR in Line 16, the loop in Lines 13 to 15 must successfully execute for all v such that all edges are embedded while maintaining the planarity of $\tilde{G}$. On the other hand, if G is nonplanar, then Theorem 6.23 guarantees that, for some step v, an unembedded back edge will be found by the loop in Lines 13 to 15, at which time the embedding structure $\tilde{G}$ will be in the embedding invariant state and contain a biconnected component on which a Condition 1 or 2 failure occurred. □

Theorem 7.2 *The procedure PlanarityTest runs in $O(n)$ time.*

Proof. The procedure begins with an invocation of Initialize in Line 1, which is $O(n)$ by Lemma 3.2.

Corollary 1.5 establishes an $O(n)$ limit on the number of edges in the input graph, whether or not it is planar. Thus, each pair of loops in PlanarityTest that iterate through all of the edges of each vertex execute the inner loop body a total of $O(n)$ times. This analysis is applicable to the following pairs of loops: Lines 2 and 3, Lines 5 and 7, Lines 5 and 9, and Lines 5 and 13.

The inner loop starting at Line 3 simply invokes procedure EmbedTreeEdge, which is $O(1)$. Thus, the total cost of Lines 2 to 4 is $O(n)$.

The inner loop starting at Line 7 simply invokes procedure Walkup, which is $O(1)$ by Lemma 5.3. Thus, the total cost of the loop in Lines 7 and 8 is $O(n)$, which is $O(1)$ amortized time per iteration of the outer loop in Line 5.

Similarly, the inner loop starting at Line 9 contains constant-time operations plus the invocation of procedure Walkdown, which is $O(1)$ by Lemma 6.24. Thus, the total cost of the loop in Lines 9 to 12 is $O(n)$, which is $O(1)$ amortized time per iteration of the outer loop in Line 5.

Likewise, the inner loop starting at Line 13 contains only constant time operations. Thus, the total cost of the loop in Lines 17 to 19 is $O(n)$, which is $O(1)$ amortized time per iteration of the outer loop in Line 5.

Therefore, the procedure PlanarityTest consists of an $O(n)$ cost from calling Initialize in Line 1, an $O(n)$ total cost from the loop over Lines 2 to 4, an $O(n)$ total cost from the loop over Lines 5 to 15, and a constant time cost for Line 16. □

8. New Planar Embedding Algorithm

This chapter covers the post-processing steps that can be performed after the operation of the procedure PlanarityTest in Chapter 7 to recover a combinatorial planar embedding of the input graph if it is planar. Section 8.1 discusses a procedure that imposes a consistent orientation to all vertices in each biconnected component in the embedding structure. Section 8.2 discusses the procedure PlanarEmbed, which creates the combinatorial planar embedding if the input graph is planar.

8.1 Orienting the Vertices

The OrientVertices defined in Figure 8.1 processes each biconnected component B in $\tilde{G}$, imparting an adjacency list order to all parent copy vertices that is consistent with the orientation of the root copy vertex in B. The procedure is called by the procedure PlanarEmbed discussed in Section 8.2 as well as by the Kuratowski subgraph isolator described in Chapter 9.

The correctness of procedure OrientVertices is stated in Lemma 8.1. The runtime analysis for procedure OrientVertices appears in Lemma 8.2.

Lemma 8.1 *Given an embedding $\tilde{G}$ in the embedding invariant state (Definition 3.8), the procedure OrientVertices converts $\tilde{G}$ to the oriented embedding invariant state (Definition 3.9).*

Proof. The algorithm iterates through each biconnected component B in $\tilde{G}$ using Lines 3 and 4. Assuming a baseline orientation given by the root copy vertex in B, the orientation of a parent copy vertex is inverted if there is an odd number of -1 signs assigned to the DFS tree edges along the ancestral DFS tree path from w to the root of the biconnected component containing w because this represents an odd number of inversions that would occur if we simply inverted w every time its nearest ancestor root copy was inverted during a merge operation (see procedure MergeBicomps in Chapter 6). □

Figure 8.1: Orienting the Vertices in the Embedding Structure

Procedure: OrientVertices
this: Embedding Structure $\tilde{G}$ resulting from procedure PlanarityTest
out: Embedding Structure $\tilde{G}$ in oriented embedding invariant state

```
(1) Delete the short-circuit edges
(2) Clear the stack S in G̃
(3) for r' from n to 2n − 1
(4)     if the parentLink of r' is not nil
(5)       Push (r', 1) onto stack S
(6)       while the stack S is not empty
(7)             Pop (w, sign) from the stack S
(8)             if sign is -1, InvertVertex(w)

(9)             for each edge record e in w's adjacency list
(10)              if e is a 'Tree Edge' to a DFS Child
(11)                Push (w, sign * (sign of e)) onto stack S
(12)                Assign 1 to sign of e
```

Lemma 8.2 *The OrientVertices procedure runs in $O(n)$ time.*

Proof. Line 1 requires a simple loop to iterate the edge array E of $\tilde{G}$, which is $O(n)$ due to the edge limits imposed by Corollary 1.5. Line 2 is constant time.

The innermost loop in Lines 9 to 12 has a constant time loop body that iterates the adjacency list of each vertex. Thus, the total cost is $O(m)$, which is $O(n)$ due to Corollary 1.5. Likewise, the invocation of InvertVertex is $O(d)$ by Lemma 4.16, but the total cost over all iterations of the loop in Line 5 is $O(m)$, which again is $O(n)$ due to Corollary 1.5. Since all other operations in the Line 6 loop are constant time, and the loop iterates over all root copy and parent copy vertices, the total cost is $O(n)$, which is $O(1)$ amortized time per vertex.

The outermost loop initiated in Line 3 iterates n times. Lines 4 and 5 are constant time, and the loop initiated in Line 6 has already been shown to achieve $O(1)$ amortized time per vertex. Thus, the procedure OrientVertices is $O(n)$. □

8.2 Obtaining a Combinatorial Planar Embedding

The procedure PlanarEmbed defined in Figure 8.2 is the functional entry point of the planar embedding algorithm. It can be called independently, but if the input graph is non-planar, the procedure IsolateKuratowskiSubgraph described in Chapter 9 must be invoked separately.

Figure 8.2: The Edge Addition Planar Embedding Algorithm

Procedure: PlanarEmbed
in: Graph G
out: An indication of whether G is planar or non-planar and the embedding structure $\tilde{G}$ in the embedding invariant state for step v if G is non-planar or a combinatorial planar embedding if G is planar

(1) (result, $\tilde{G}$) $\leftarrow$ PlanarityTest(G)
(2) if result is PLANAR
(3) $\quad\tilde{G}$.OrientVertices()
(4) $\quad\tilde{G}$.JoinBicomps()

(5) return (result, $\tilde{G}$)

The correctness of procedure PlanarEmbed is established by Lemma 8.3. The run-time analysis for procedure PlanarEmbed appears in Lemma 8.4.

Lemma 8.3 *On an input graph G with at most kn edges for some constant $k \geq 3$, the procedure PlanarEmbed produces a combinatorial planar embedding of G in standard adjacency list format. For non-planar graphs, the PlanarEmbed procedure yields the embedding structure $\tilde{G}$ in embedding invariant state (Definition 3.8) containing a biconnected component on which a Condition 1 or 2 failure occurred.*

Proof. If the graph is non-planar, then Line 2 prevents alteration of the embedding structure $\tilde{G}$ resulting from the PlanarityTest invocation, which meets the stated requirements according to Theorem 7.1. For planar graphs, the procedure OrientVertices is invoked, which transforms $\tilde{G}$ to the oriented embedding invariant state according to Lemma 8.1. By Lemma 4.20, the invocation of JoinBicomps then merges the

root copy of each biconnected component with its parent copy without any change of vertex orientation. If the parent copy is degree 0 (indicating a DFS tree root), then the adjacency list of the root copy is transferred to the parent copy. Otherwise, the orientation of the parent copy of the cut vertex is preserved after the merge because the edges incident to a root copy are inserted without inversion between two edges of the parent copy. Thus, $\tilde{G}$ is a combinatorial planar embedding of the input graph in standard adjacency list format because it contains no root copy vertices and one parent copy vertex per vertex of the input graph, the adjacency list of each parent copy vertex contains no short-circuit edges and all incident edges corresponding to those in the input graph (by Theorem 7.1), and the parent copy vertices all have a consistent adjacency list orientation. □

Lemma 8.4 *The PlanarEmbed procedure runs in $O(n)$ time.*

Proof. Line 1 performs a simple assignment ($\tilde{G}$ is handled by reference) and also invokes the procedure PlanarityTest, which is $O(n)$ by Theorem 7.2. Line 2 is a constant time test and branch operation. Line 3 invokes OrientVertices, which is $O(n)$ by Lemma 8.2. Line 4 invokes JoinBicomps, which is $O(n)$ by Lemma 4.21. Line 5 is constant time. □

9. New Kuratowski Subgraph Isolator

The process of isolating a Kuratowski subgraph of the input graph G is performed on the embedding structure $\tilde{G}$ that results from an invocation of the procedure PlanarityTest in Chapter 7. The process begins by choosing one of the non-planarity minors identified in the lemmas leading to the proof of correctness of the procedure Walkdown (Theorem 6.23 in Chapter 6). These non-planarity minors appear in Figures 6.3(a), 6.4, 6.5, 6.6, and 6.7.

Section 9.1 defines an augmentation of the embedding structure to store additional variables pertaining to the selection of a non-planarity minor, and it defines the states entered by the augmented embedding structure during the selection process. Section 9.2 provides simple, low-level procedures for populating most of the variables in the augmented data structure, and Section 9.3 provides procedures for finding (and marking) the internal paths appearing in the latter three non-planarity minors. Section 9.4 provides the top-level routine that selects a non-planarity minor based on the results of the low-level routines and the characteristics of the internal paths found in the embedding.

Once the type of non-planarity minor is selected, it is used to choose a procedure for isolating a Kuratowski subgraph. Section 9.5 defines an additional embedding structure augmentation, and some additional low-level procedures are defined in Section 9.6 to perform tasks such as marking desired paths and adding unembedded edges to completely form a Kuratowski subgraph. The procedures defined in Sections 9.7 through 9.11 use the low-level operations to mark paths in the embedding structure $\tilde{G}$ that correspond to the vertices and edges in the five non-planarity minors. The final step is to delete all unmarked vertices and edges, yielding in $\tilde{G}$ a Kuratowski subgraph of G as the final result of the top-level procedure IsolateKuratowskiSubgraph described in Section 9.12.

9.1 Data Structure Context for Choosing a Non-planarity Minor

Immediately after the execution of procedure PlanarityTest, the embedding structure $\tilde{G}$ is in the non-planar embedding invariant state given by Definition 9.1. To simplify a number of procedures, the first step of selecting a non-planarity minor is to invoke procedure OrientVertices, which results in $\tilde{G}$ having the oriented non-planar embedding invariant state given in Definition 9.2. Also to simplify a number of procedures, the variables listed in Table 9.1 are added to the embedding structure and used to store information relevant to selecting a non-planarity minor. Based in part on setting these new variables, Definitions 9.4, 9.5, and 9.6 describe the states entered by the augmented embedding structure during the non-planarity minor selection process.

Definition 9.1 *The embedding is in the* non-planar embedding invariant state *if it in the embedding invariant state and it contains a root copy of v in a biconnected component B on which a Condition 1 or 2 Walkdown termination occurred.*

Definition 9.2 *The embedding is in the* oriented non-planar embedding invariant state *if it in the non-planar embedding invariant state, there are no short-circuit edges, and the sign of all edge records is 1.*

Definition 9.3 *Given r', x, y and w from Table 9.1, the following paths are defined:*

1. *The* RXW path *is the external face path (r', ..., x, ..., w), excluding r' and w (and their incident edges).*

2. *The* high RXW path *is the path contained by the RXW path between and excluding r' and x.*

3. *The* low RXW path *is the path contained by the RXW path from x to w, excluding w.*

4. *The* RYW path *is the external face path (r', ..., y, ..., w), excluding r' and w (and their incident edges).*

Table 9.1: Augmentation To Embedding Structure for Storing Non-planarity Minor Selection Context

Variable	Description
G	A reference to the input graph.
minorType	Initially empty, but takes on the value A, B, C, D or E once the characteristics of the corresponding non-planarity have been found in the embedding.
r'	The root copy of the biconnected component of central interest in the non-planarity minor. It is the root copy of v on which the Walkdown failed with Condition 2, except for minor A, which requires the root copy that is on top of the stack when the Walkdown fails with Condition 1.
x	The stopping vertex that can be reached by traversing the external face path from r' that starts with the edge indicated by link[0] in r'.
y	The stopping vertex that can be reached by traversing the external face path from r' that starts with the edge indicated by link[1] in r'.
w	A pertinent vertex on the lower path between and excluding x and y along the external face of the biconnected component with root r'.
p_x	Initially *nil*, this variable is set for minors C, D, and E to indicate the point of attachment along the external face path $(r', \ldots, x, \ldots, w)$ of the highest x-y path.
p_y	Initially *nil*, this variable is set for minors C, D, and E to indicate the point of attachment along the external face path $(r', \ldots, y, \ldots, w)$ of the highest x-y path.
z	Initially *nil*, this variable is set for minors D and E. Note that z is not set for minors B and C because it is not necessary to know z to determine that minor B or C has occurred. For minor D, z indicates a vertex on the x-y path distinct from p_x and p_y that is the endpoint of a second path p of the form $(r', \ldots, z)$, where all vertices in the path (except r') are embedded inside the biconnected component rooted by r'. For minor E, z indicates an externally active vertex (distinct from p_x and p_y) on the lower external face path $(p_x, \ldots, w, \ldots, p_y)$.

5. *The* high RYW path *is the path contained by the RYW path between and excluding r' and y.*

6. *The* low RYW path *is the path contained by the RYW path from y to w, ex-*

cluding w.

Definition 9.4 *The embedding structure $\tilde{G}$ is in the* non-planarity minor initial state *if it is in the oriented embedding invariant state with the following exceptions:*

1. *The variables G, r', x, y and w in $\tilde{G}$ are set in the manner described in Table 9.1.*
2. *The variables p_x, p_y, and z in $\tilde{G}$ are set to nil.*
3. *The variable minorType in $\tilde{G}$ is empty.*
4. *The visited members of all vertices and edges are equal to 0.*
5. *Except for r' and w, the type member of each vertex structure on the external face of the biconnected component with root r' are modified as follows:*
   (a) *Vertices on the high RXW path have the type high RXW.*
   (b) *Vertices on the low RXW path have the type low RXW.*
   (c) *Vertices on the high RYW path have the type high RYW.*
   (d) *Vertices on the low RYW path have the type low RYW.*

   Note that all internal vertices in the biconnected component rooted by r' are easily identifiable because they receive no type designation and are not equal to r' or w.

Definition 9.5 *The embedding structure $\tilde{G}$ is in the* non-planarity minor x-y path found state *if it is in the non-planarity minor initial state with the following exceptions:*

1. *The visited members of vertex structures and edge records along the highest x-y path are equal to 1.*
2. *The points of attachment of the highest x-y path are indicated by the variables p_x and p_y in $\tilde{G}$ in the manner described in Table 9.1.*

Definition 9.6 *The embedding structure $\tilde{G}$ is in the* non-planarity minor found state *if it is in the non-planarity minor initial state with the following exceptions:*

1. *The variable minorType is set in the manner described in Table 9.1.*
2. *If minorType is C, D or E, then $\tilde{G}$ has the two exceptional settings listed in Definition 9.5.*
3. *If minorType is C, D or E, then z is set in the manner described in Table 9.1.*
4. *If minorType is D, then the visited members of vertex structures and edge records are equal to 1 along the internal path $(r', \ldots, z)$ described in Table 9.1 under the entry for z.*

9.2 Low Level Operations for Choosing a Non-planarity Minor

This section presents an overview of several low level procedures used to help select a non-planarity minor. The procedures are described in prose rather than by pseudo-code since, by this point, the reader is assumed to be sufficiently familiar with our data structures that these simple utility procedures should present no difficulties. Note that when these procedures are called, the embedding structure $\tilde{G}$ is passed implicitly as the 'this' parameter (as defined in Section 3.1).

The procedure *InitializeNonplanarityMinorContext* receives the embedding structure $\tilde{G}$ in the oriented non-planar embedding invariant state (Definition 9.2) as well as the input graph G. The procedure transforms $\tilde{G}$ to the non-planarity minor initial state (Definition 9.4). Much of the defined state consists of trivial assignments and assignment loops, except for setting r', x, y and w. To begin finding r', an unembedded edge (v, z) is located by searching in $\tilde{G}$ the vertices that, in G, are neighbors of v to find a vertex z with an adjacentTo member equal to v. Then, we follow the DFS tree ancestor path from z to a DFS child c of v, then obtain $v' = c + n$. Next, we traverse the link[0] and link[1] edges leading from v' to obtain the first active vertices x and y along each path. If both x and y are stopping vertices, then the desired

root r' is equal to v', and the stopping vertices are the desired values for x and y. On the other hand, if either x or y is not a stopping vertex, then the Walkdown procedure terminated after descending to but without merging some pertinent child biconnected component (see Condition 1 in Chapter 6). We invoke the Walkdown on v' to reconstruct the stack S of pertinent biconnected component roots that lead to the Condition 1 failure of Walkdown. The desired root r' is the root of the pertinent child biconnected component in the topmost 2-tuple of integers on the stack S. The link[0] and link[1] external face paths from r' lead to the desired stopping vertices x and y. Once r', x and y have been identified, the pertinent vertex w if obtained using a simple loop to search the lower external face path between (and excluding) x and y.

The procedure *HideInternalEdgesOfRoot* receives the embedding structure $\tilde{G}$ in the non-planarity minor initial state (Definition 9.4). The procedure temporarily deletes the edges incident to r' except the two edges incident to r' along the external face. For each edge record e in the adjacency list of r' except the two indicated by link[0] and link[1] in r', this procedure performs the following actions. The edge record e is pushed onto the stack S in $\tilde{G}$. The edge record e_{twin} is obtained using the twinLink in edge record e. A standard $O(1)$ doubly linked list delete is used to omit e and e_{twin} from the adjacency lists containing them. However, the edge records e and e_{twin} are not removed from the edge record array E in $\tilde{G}$, and the information they contain is not altered.

The procedure *PopAndUnmarkVerticesAndEdges* receives the embedding structure $\tilde{G}$ in the non-planarity minor initial state (Definition 9.4) except for the effects of procedure HideInternalEdgesOfRoot described above and except for having the value 1 in the visited members of some vertex structures and edge records. Additionally, the procedure receives a vertex s, and the stack S in $\tilde{G}$ is assumed to contain zero or more 2-tuples consisting of a vertex structure reference on top of an edge record reference, under which are zero or more other edge record references not to be perturbed by this procedure (pushed by procedure HideInternalEdgesOfRoot described above). A simple loop is performed that terminates if the stack becomes empty, if

the top element of the stack refers to an edge record, or if vertex s appears on the top of the stack. If none of the terminating conditions occurs, then the loop body pops a vertex i then an edge record e, and it clears (assigns 0 to) the visited members of i, e and the edge record e_{twin} obtained using the twinLink of e.

The procedure *RestoreInternalEdgesOfRoot* receives the embedding structure $\tilde{G}$ in the non-planarity minor x-y path found state (Definition 9.5), except for the effects of procedure HideInternalEdgesOfRoot described above. While the stack is not empty, the procedure pops an edge record e from the stack S in $\tilde{G}$, and it computes e_{twin} using the twinLink in e. Then, the link[0] and link[1] members of edge records e and e_{twin} are used to reinsert them into their respective adjacency lists in $O(1)$ time. This reverses the action of the procedure HideInternalEdgesOfRoot, and puts the embedding structure $\tilde{G}$ in the non-planarity minor x-y path found state.

The procedure *FindExtActivityBelowXYPath* receives the embedding structure $\tilde{G}$ in the non-planarity minor x-y path found state (Definition 9.5 with established the points of attachment p_x and p_y of an x-y path). The x-y path is assumed to be a low x-y path. Along the lower external face path between and excluding p_x and p_y (the path that excludes r'), the procedure searches for an externally active vertex z by invoking VertexActiveStatus (see Section 4.4) on each vertex. The vertex z is returned, or *nil* is returned if there are no externally active vertices along the path explored.

9.3 Identifying Key Paths within a Biconnected Component

Non-planarity minors C, D and E in Figures 6.5, 6.6, and 6.7 include an x-y path through the biconnected component B with the root r'. Non-planarity minor D also contains a path of vertices and edges from r' to a vertex z along the x-y path; other than r', the vertices and edges on this path are embedded internally within B. This section covers two procedures for identifying these paths by setting the 'visited' members of their representative vertex structures and edge records.

The procedure MarkHighestXYPath defined in Figure 9.1 receives the embedding structure $\tilde{G}$ in non-planarity minor initial state. The procedure MarkHighestXY-

Path uses the 'visited' member of vertices and edges in $\tilde{G}$ to mark the highest x-y path with points of attachment p_x and p_y. The procedure is not called unless an x-y path is guaranteed to exist.

The correctness of procedure MarkHighestXYPath is established by Lemma 9.1. The run-time analysis for procedure MarkHighestXYPath appears in Lemma 9.2.

Lemma 9.1 *Given an embedding structure $\tilde{G}$ in the non-planarity minor initial state (Definition 9.4) and containing an x-y path in the biconnected component rooted by r', the MarkHighestXYPath procedure transforms $\tilde{G}$ to the non-planarity minor x-y path found state (Definition 9.5).*

Proof. Lines 1 and 2 temporarily remove all edges incident to r' except the two along the external face, storing references to the removed edges on stack S. Thus, r' becomes degree 2. The bounding walk of the resulting proper face containing r' cannot contain the w candidate since this would violate the assumption that an x-y path is known to exist in the biconnected component. This bounding walk is performed by the loop in Lines 5 to 19. Portions of the bounding walk whose endpoints are both on the RXW path are omitted from the x-y path by Lines 13 to 15 such that the last visited vertex on the RXW path is the first vertex p_x marked on the x-y path. The first visited vertex p_y on the RYW path is the last vertex marked on the x-y path since the loop terminates when p_y is found. The edges and vertices marked form an x-y path because portions of the bounding walk corresponding to separable components are omitted from the x-y path by Line 11. Moreover, the assumption that the x-y path obtained by this procedure is not the highest x-y path contradicts the fact that the x-y path was obtained by traversing the bounding walk of the proper face containing r'. Thus, since the loop marks the highest x-y path and identifies p_x, Line 20 sets p_y, and Lines 21 to 23 restore the internal edges of r', the desired state given by Definition 9.5 is achieved. □

Lemma 9.2 *The MarkHighestXYPath procedure runs in $O(n)$ time.*

Figure 9.1: Marking the Highest x-y Path

Procedure: MarkHighestXYPath
this: Embedding structure $\tilde{G}$ in non-planarity minor initial state
out: Embedding structure $\tilde{G}$ in non-planarity minor x-y path found state

(1) Clear the stack S
(2) HideInternalEdgesOfRoot()

(3) Set s equal to r'
(4) Set e equal to the link[1] member of r'
(5) while the type of s is neither 'high RYW' nor 'low RYW'
(6) Set e equal to link[1] of e
(7) if e indicates a vertex structure
(8) Set e equal link[1] of e
(9) Set s equal to the neighbor member of e
(10) Set e equal to the twinLink of e

(11) if s has been visited, PopAndUnmarkVerticesAndEdges(s)
(12) else
(13) if the type of s is either 'high RXW' or 'low RXW'
(14) Set p_x equal to s
(15) PopAndUnmarkVerticesAndEdges(nil)

(16) Set visited member of s equal to 1
(17) if $s \neq p_x$
(18) Assign 1 to the visited members of e and its twin edge record

(19) Push e then s onto stack S

(20) Set p_y equal to s

(21) while stack S is non-empty and the top element indicates a vertex
(22) Pop the top two elements from stack S
(23) RestoreInternalEdgesOfRoot()

Proof. Line 1 is typically $O(1)$. Line 2 invokes procedure HideInternalEdgesOfRoot, which is $O(d)$ where d is the degree of r' and is less than n. Lines 3 and 4 are constant time assignments. Lines 5 to 19 form a loop with a constant amortized time loop body (all operations are constant, except the invocations of PopAndUnmarkVerticesAndEdges, the cost of which can be associated with prior iterations of the loop body). Thus, the loop is $O(n)$ since it iterates once per vertex on the bounding walk of the proper face containing r'. Line 20 is a constant time assignment. Lines 21 and 22 pop a stack whose length is commensurate with the length of the x-y path, which is worst-case $O(n)$. Finally, Line 23 invokes RestoreInternalEdgesOfRoot, which is $O(n)$ for the reason given above for Line 2. □

The procedure *MarkVtoZPath* defined in Figure 9.2 receives the embedding structure $\tilde{G}$ in the non-planarity minor x-y path found state in which the identified x-y path is a low x-y path. The procedure searches for a path p of the form $(r', \ldots, z)$, where all vertices (except r') are inside B and z is the only vertex in both p and in the x-y path. If the desired path p is found, then the 'visited' members are marked in the representative vertex structures and edge records of p, and the variable z in $\tilde{G}$ is set to a non-*nil* value.

The correctness of procedure MarkVtoZPath is established by Lemma 9.3. The run-time analysis for procedure MarkVtoZPath appears in Lemma 9.4.

Lemma 9.3 *Given an embedding structure $\tilde{G}$ in the non-planarity minor x-y path found state (Definition 9.5) and containing a low x-y path in the biconnected component B rooted by r', the MarkVtoZPath procedure returns with $\tilde{G}$ in the same state except that, if there exists a path $p = (r', \ldots, z)$ with all vertices (except r') inside B, then z is recorded in $\tilde{G}$ and the visited members of the vertex structures and edge records representing p are set equal to 1.*

Proof. Since B is biconnected, it contains no cut vertices. However, the procedure MarkHighestXYPath obtains the x-y path by first removing the internal edges incident to r', which can result in cut vertices. When MarkHighestXYPath encounters a vertex s already visited, it unmarks all vertices and edges visited since its last

Figure 9.2: Marking the v-z Path

Procedure: MarkVtoZPath
this: Embedding structure $\tilde{G}$ in non-planarity minor x-y path found state
with an identified x-y path that is a low x-y path.
out: Embedding structure $\tilde{G}$ in non-planarity minor x-y path found state
except if the desired path p is found, then the visited members are assigned 1
in the vertex structures and edge records representing path p and the vertex
in both p and the x-y path is assigned into z in $\tilde{G}$

(1) Let e be the edge record in the adjacency list of p_x with visited member equal to 1
(2) while the visited member of e is not equal to 0
(3) Set e_{twin} equal to the twinLink of e
(4) Set e equal to the link[1] member of e_{twin}
(5) if e indicates a vertex structure, set e equal to the link[1] member of e

(6) Set e_{twin} equal to the twinLink of e
(7) Set s equal to the neighbor member of e_{twin}
(8) if $s \neq p_y$
(9) Set z equal to s
(10) while $s \neq r'$
(11) Set s equal to the neighbor member of e

(12) Set the visited members of s, e and e_{twin} equal to 1

(13) Set e equal to the link[1] member of e_{twin}
(14) if e indicates a vertex structure, set e equal to the link[1] member of e
(15) Set e_{twin} equal to the twinLink of e

encounter with s. Thus, if the link[1] successor edge of e was not marked during the execution of MarkHighestXYPath, then s is either directly adjacent to r' by an internal edge or s becomes a cut vertex when the internal edges incident to r' are removed.

The MarkVtoZPath procedure traverses the x-y path starting with the first internal vertex neighbor of p_x. If a vertex s meeting the criterion above is found before p_y is encountered, then s is assigned to z in $\tilde{G}$ because the start of the desired path p has been found. The vertex structures and edge records representing p are marked by simply repeating the traversal of the (possibly empty) portion of the walk

performed by MarkHighestXYPath immediately after it first encounters s. However, since the internal edges incident to r' have not been removed, the traversal proceeds to r' the first time it encounters a neighbor of r'.

Inversely, if there is no vertex s meeting the above criterion, then there exists no desired path p. If in the sequence of vertices and edges of the x-y path, the succeeding edge of each internal vertex is the link[1] successor edge of the vertex's preceding edge, then the entire x-y path forms part of the bounding cycle of a single proper face that also includes r'. By contradiction, a desired path p attaches to the x-y path at some point z such that the preceding and succeeding edges of z along the x-y path appear in two separate proper faces that also include r'. □

Lemma 9.4 *The MarkVtoZPath procedure runs in $O(n)$ time.*

Proof. Line 1 contains an implied simple loop, which is $O(n)$ since it performs constant work in the loop body and its iteration is limited by the degree of p_x. Lines 2 to 5 traverse the x-y path, performing constant work per vertex and edge, for a total worst case of $O(n)$. Lines 6 to 9 perform constant time operations. Lines 10 to 15 perform a loop that marks the vertices and edges of the desired path p. The loop achieves $O(n)$ time since p is a path and the loop body contains only constant time operations. □

9.4 Choosing a Non-planarity Minor

The procedure ChooseTypeOfNonplanarityMinor defined in Figure 9.3 identifies a non-planarity minor that can be used as the basis of isolating a Kuratowski subgraph. This is accomplished by testing for the existence of the additional representative structure in $\tilde{G}$ beyond the subgraph and unembedded edges known to exist by virtue of premature termination of the Walkdown procedure due to Condition 1 or 2 (see Chapter 6). The procedure selects a minorType of A, B, C, D or E depending on whether it recognized the additional structure for the minor in Figure 6.3(a), Figure 6.4, Figure 6.5, Figure 6.6 or Figure 6.7, respectively. The subordinate

Figure 9.3: Choosing a Non-planarity Minor

```
Procedure: ChooseTypeOfNonplanarityMinor
this: Embedding structure G̃ in the non-planar embedding invariant state
in: The input graph G
out: Embedding structure G̃ in the non-planarity minor found state

(1) OrientVertices()
(2) InitializeNonplanarityMinorContext(G)

(3) if r' is not a root copy of v
(4)    Assign 'A' to minorType

(5) else if the pertinentBicompList of w is not empty
(6)         Set w' equal to the value of the last element of pertinentBicompList of w
(7)         Set c equal to w'-n
(8)         if the lowpoint of c is less than v
(9)            Assign 'B' to minorType

(10) if minorType has an empty value (was not assigned 'A' or 'B')
(11)    MarkHighestXYPath()

(12)    if the type of p_x or p_y is 'high RXW'
(13)       Assign 'C' to minorType
(14)    else MarkVtoZPath()
(15)         if z ≠ nil, Assign 'D' to minorType
(16)         else Set z equal to the result of FindExtActivityBelowXYPath()
(17)              Assign 'E' to minorType
```

procedures invoked by ChooseTypeOfNonplanarityMinor make other informational modifications such that the embedding $\tilde{G}$ is transformed to the non-planarity minor found state.

The correctness of procedure ChooseTypeOfNonplanarityMinor is established by Lemma 9.5. The run-time analysis for procedure ChooseTypeOfNonplanarityMinor appears in Lemma 9.6.

Lemma 9.5 *Given an input graph G for which the procedure PlanarityTest produced the embedding structure $\tilde{G}$ in the non-planar embedding invariant state (Defi-*

nition 9.1), the ChooseTypeOfNonplanarityMinor procedure transforms $\tilde{G}$ to the non-planarity minor found state (Definition 9.6).

Proof. Lines 1 and 2 transform $\tilde{G}$ to the non-planarity minor initial state (Definition 9.4), after which the test for minor A follows directly from Lemma 6.13. Line 5 determines whether w has a pertinent child biconnected component, and Lines 6 to 8 test whether it is externally active, in which case the choice of minor B follows from Lemma 6.17. If neither minor A nor B were selected (Line 10), then the biconnected component B with root r' (a root copy of v) must contain an x-y path according to Theorem 6.23. By Lemma 9.1, the invocation of MarkHighestXYPath() in Line 13 transforms $\tilde{G}$ to the non-planarity minor x-y path found state (Definition 9.5), after which the test for minor C in Lines 12 and 13 follows directly from Lemma 6.18. If the x-y path is a low x-y path, then execution proceeds to Lines 14 and 15. If the procedure MarkVtoZPath marks the vertex structures and edge records of an internal path p and obtains the vertex z common to p and the x-y path as described in Lemma 9.3, then the selection of minor D is appropriate according to Lemma 6.19. On the other hand, if p is not found by the procedure MarkVtoZPath, then Theorem 6.23 guarantees that there is an externally active vertex z distinct from p_x and p_y on the external face path $(p_x, \ldots, w, \ldots, p_y)$. The vertex z is obtained on Line 16 such that the selection of minor E is appropriate according to Lemma 6.20. Thus, in all cases, $\tilde{G}$ enters the non-planarity minor found state. □

Lemma 9.6 *The ChooseTypeOfNonplanarityMinor procedure runs in $O(n)$ time.*

Proof. All operations are constant time except the procedure invocations. Each procedure is invoked once. OrientVertices is $O(n)$ by Lemma 8.2. MarkHighestXYPath and MarkVtoZPath are each $O(n)$ by Lemmas 9.2 and 9.4. The remaining low-level procedures contain simple $O(n)$ loops. □

9.5 Data Structure Context for Isolating a Kuratowski Subgraph

Immediately after the execution of procedure ChooseTypeOfNonplanarityMinor, the embedding structure $\tilde{G}$ is in the non-planarity minor found state given by Definition 9.6. To simplify a number of procedures, the variables listed in Table 9.2 are added to the embedding structure and used to store information relevant to isolating a Kuratowski subgraph. Based in part on setting these new variables, Definitions 9.7, 9.8 and 9.9 describe the states entered by the augmented embedding structure during the Kuratowski subgraph isolation process.

Definition 9.7 *The embedding structure $\tilde{G}$ is in the* isolator initial state *if it is in the non-planarity minor found state with the exception that the additional variables u_x, d_x, u_y, d_y, d_w, u_z, and d_z are added to $\tilde{G}$ and set in the manner described in Table 9.2.*

Definition 9.8 *The embedding structure $\tilde{G}$ is in the* isolation in progress state *if it is in the isolator initial state except that zero or more additional vertex structures and edge records have a visited member set to 1.*

Definition 9.9 *The embedding structure $\tilde{G}$ is in the* Kuratowski subgraph marked state *if it represents a subgraph of G using only parent copy vertex structures and edge records, and if the subgraph of $\tilde{G}$ consisting only of vertex structures and edge records with a visited member equal to 1 forms a $K_{3,3}$ or K_5 homeomorph.*

9.6 More Low Level Operations for Isolating Kuratowski Subgraphs

This section presents an overview of several more low level procedures required by the Kuratowski subgraph isolator. As in Section 9.2, the procedures are described in prose rather than by pseudo-code. Note that when these procedures are called, $\tilde{G}$ is passed implicitly as the 'this' parameter (as defined in Section 3.1).

Table 9.2: Augmentation To Embedding Structure for Storing Kuratowski Subgraph Isolation Context

Variable	Description
u_x, d_x	The endpoints of a back edge in G, not in $\tilde{G}$, and represented by the edge (u, x) in the non-planarity minors. If the leastAncestor of x is less than v, then u_x is equal to the leastAncestor of x, and d_x is equal to x. Otherwise, u_x equals the lowpoint of the first child c in the separatedDFSChildList of x, and d_x is computed by obtaining 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). In this case, (u, x) represents the back edge (u_x, d_x) plus the DFS tree path from d_x to x.
u_y, d_y	The endpoints of a back edge in G, not in $\tilde{G}$, and represented by the edge (u, y) in the non-planarity minors. The values of u_y and d_y are computed using y in the same manner as u_x and d_x are computed using x.
d_w	The descendant endpoint of the back edge (v, d_w) in G, not in $\tilde{G}$, and represented by the edge (v, w) in the non-planarity minors. If the adjacentTo member of w equals v and the minorType is not B, then d_w equals w. Otherwise, d_w is computed by obtaining the root w' of the last pertinent child biconnected component of w, calculating $c = w' - n$, then obtaining the neighbor of v in G with the least DFI greater than or equal to c (such that d_w is in the DFS subtree rooted by c). Note that w' could be the root of any pertinent child biconnected component, except that the biconnected component must be externally active for minor B.
u_z, d_z	These are *nil* unless minorType is B or E. For minor B, u_z is equal to the lowpoint of the same vertex c used to help obtain d_w, and d_z is computed by obtaining the neighbor of u_z in G with the least DFI greater than or equal to c. For minor E, the values of u_z and d_z are computed using z in the same manner as u_x and d_x are computed using x.

The procedure *InitializeIsolatorContext* receives the embedding structure $\tilde{G}$ in the non-planarity minor found state (Definition 9.6) and transforms $\tilde{G}$ to the isolator initial state (Definition 9.7). This involves obtaining values for the additional variables in Table 9.2, which also describes the methods for computing these values.

The procedure *MarkPathAlongBicompExtFace* receives the embedding structure $\tilde{G}$ in the isolation in progress state and two vertices s and t, both on the external face of the biconnected component B rooted by the vertex r'. Vertex s can be any

vertex on the RXW path or r', and t can be any vertex on the RYW path or r'. The procedure begins with vertex s, proceeds around the RXW path to w, then from w along the RYW path to t. All vertices and edges encountered are marked by setting the 'visited' members of all the representative vertex structures and edge records equal to 1. Thus, if s and t are equal to r' then all of the vertices and edges on the external face of B are marked. Also, if r' is marked by this procedure, then the parent copy of r' is not marked because it will be marked in minor A when the DFS tree path from r to v and in all other minors when the edge (v, d_w) is embedded.

The procedure *MarkDFSPath* receives the embedding structure $\tilde{G}$ in the isolation in progress state, a vertex a and a vertex d, which is a descendant of a. If d is a root copy, then set d equal to the vertex indicated by its parentLink. This procedure assigns 1 to the 'visited' members of the vertex structures and edge records along the DFS tree path from d to a, jumping from root copy to parent copy as necessary. First, the visited member in the parent copy of d is set equal to 1. Then, the procedure uses a simple loop whose body obtains from the adjacency list of d the tree edge indicating a root copy or parent copy of the DFSParent p of d. The visited members are set equal to 1 in the edge records of the tree edge and the vertex structure for the parent copy of p (it is unnecessary to mark root copies, which will be eliminated in the final result).

The procedure *MarkDFSPathsToDescendants* receives the embedding structure $\tilde{G}$ in the isolation in progress state. For each of the unembedded edges (u_x, d_x), (u_y, d_y), and (v, w) as well as the edge (u_z, d_z) if d_z is non-*nil*, this procedure invokes the procedure MarkDFSPath to mark the corresponding tree path $(x, \ldots, d_x)$, $(y, \ldots, d_y)$, and $(w, \ldots, d_w)$ as well as $(w, \ldots, d_z)$ if d_z is non-*nil*. Note that d_z is only non-*nil* for minors B and E, and z is an unidentified descendant of w in minor B while $w = z$ for the cases in minor E where the path $(w, \ldots, d_z)$ must be marked (hence, we use w instead of z as the ancestor in the path to be marked). For each invocation of MarkDFSPath, the parameter a is assigned the ancestor endpoint of the given path, and d is assigned the descendant. This procedure accounts for the fact that an edge in a non-planarity minor may be representative of a DFS tree path plus a back edge.

Figure 9.4: Isolating Minor A

Procedure: IsolateMinorA
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of A
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) MarkPathAlongBicompExtFace(r', r')
(2) MarkDFSPath(min(u_x, u_y), r')
(3) MarkDFSPathsToDescendants()
(4) JoinBicomps()
(5) AddAndMarkUnembeddedEdges()

For example, the edge (u, x) in a non-planarity minor may represent a direct back edge from x to an ancestor of v, or there may be a back edge to an ancestor of v from a descendant d_x of x in another biconnected component.

The procedure *AddAndMarkUnembeddedEdges* receives the embedding structure in the Kuratowski subgraph marked state except that it is missing the edges (u_x, d_x), (u_y, d_y), and (v, d_w) as well as the edge (u_z, d_z) if u_z is non-*nil*. For each of these missing edges, the procedure adds the two edge records that represent the edge to the adjacency lists of the vertex structures representing the respective endpoints. The procedure also increments m, and it assigns 1 to the visited members of the edge records and the endpoint vertex structures of each edge it adds. As a result, $\tilde{G}$ enters the Kuratowski subgraph marked state.

9.7 Isolating Minor A

The procedure IsolateMinorA defined in Figure 9.4 processes the embedding structure $\tilde{G}$ according to non-planarity minor A appearing in Figure 6.3(a). As a result, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$.

The correctness of procedure IsolateMinorA is established by Lemma 9.7. The run-time analysis for procedure IsolateMinorA appears in Lemma 9.8.

Lemma 9.7 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to A, the IsolateMinorA procedure transforms $\tilde{G}$ to the Kuratowski*

subgraph marked state.

Proof. The invocation of MarkPathAlongBicompExtFace in Line 1 assigns a 1 to the visited members of vertex structures and edge records in the external face cycle corresponding to the cycle $(r, \ldots, x, \ldots, w, \ldots, y, \ldots, r)$ in minor A, except the parent copy of r, which is marked in Line 2. The invocation of MarkDFSPath in Line 2 assigns a 1 to the visited members of vertex structures and edge records in the DFS tree path corresponding to the path $(u, \ldots, v, \ldots, r)$ in minor A. Between the addition and marking of back edges (u_x, d_x), (u_y, d_y), and (v, d_w) in Line 5, and the marking in Line 3 of the DFS tree paths connecting d_x to x, d_y to y and d_w to w, the paths in $\tilde{G}$ corresponding to the edges (u, x), (u, y), and (v, w) are marked. The invocation of JoinBicomps in Line 4 completes the correspondence with minor A such that a subgraph of G homeomorphic to $K_{3,3}$ has been marked in $\tilde{G}$. □

Lemma 9.8 *The IsolateMinorA procedure runs in $O(n)$ time.*

Proof. The procedures MarkPathAlongBicompExtFace, MarkDFSPath, and MarkDFSPathsToDescendants are simple, low-level procedures that clearly achieve $O(n)$ in the worst case. The procedure JoinBicomps is $O(n)$ by Lemma 4.21. The AddAndMarkUnembeddedEdges procedure is constant time. All procedures are invoked a constant number of times. □

9.8 Isolating Minor B

The procedure IsolateMinorB defined in Figure 9.5 processes the embedding structure $\tilde{G}$ according to non-planarity minor B appearing in Figure 6.4. As a result of the processing, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$.

The correctness of procedure IsolateMinorB is established by Lemma 9.9. The run-time analysis for procedure IsolateMinorB appears in Lemma 9.10.

Figure 9.5: Isolating Minor B

Procedure: IsolateMinorB
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of B
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) MarkPathAlongBicompExtFace(r', r')
(2) MarkDFSPath(min(u_x, u_y, u_z), max(u_x, u_y, u_z))
(3) MarkDFSPathsToDescendants()
(4) JoinBicomps()
(5) AddAndMarkUnembeddedEdges()

Lemma 9.9 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to B, the IsolateMinorB procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

Proof. The proof is the same as in Lemma 9.7 except as follows. Firstly, there are as many as three ancestors of v involved in minor B, corresponding to the external activity of x, y and z. Thus, the DFS tree path in $\tilde{G}$ that corresponds to vertex u in minor B extends from the maximum DFI to the minimum DFI of the three ancestors, which is marked in Line 2. Secondly, though vertices u and v in minor B are required to form a $K_{3,3}$, the edge (u, v) in minor B is not required. Thus, the vertices and edges are not marked in the corresponding DFS tree path in $\tilde{G}$ from v to the ancestor of maximum DFI associated with u in minor B (except, of course, that the endpoints of that path are marked by Lines 2 and 5). Thirdly, minor B contains an extra vertex z as well as the edges (w, z) and (u, z) that do not appear in minor A but which are isolated by the same methods as those used to isolate the path $(u_x, \ldots, d_x, \ldots, x)$. Finally, since r' is a root copy of v, the invocation of MarkPathAlongBicompExtFace marks the cycle in $\tilde{G}$ corresponding to the cycle $(v, \ldots, x, \ldots, w, \ldots, y, \ldots, v)$ in minor B, except the parent copy of v, which is marked when the edge (v, d_w) is added. Thus, a correspondence has been made with the portion of minor B necessary to mark in $\tilde{G}$ a subgraph of G homeomorphic to $K_{3,3}$. □

Lemma 9.10 *The IsolateMinorB procedure runs in $O(n)$ time.*

Figure 9.6: Isolating Minor C

Procedure: IsolateMinorC
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of C
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) if the type of p_x is 'high RXW'
(2) if the type of p_y is 'low RYW', $p_y \leftarrow y$
(3) MarkPathAlongBicompExtFace(r', p_y)
(4) else MarkPathAlongBicompExtFace(x, r')

(5) MarkDFSPath(min(u_x, u_y), r')
(6) MarkDFSPathsToDescendants()

(7) JoinBicomps()
(8) AddAndMarkUnembeddedEdges()

Proof. Same arguments as in Lemma 9.8. □

9.9 Isolating Minor C

The procedure IsolateMinorC defined in Figure 9.6 processes the embedding structure $\tilde{G}$ according to non-planarity minor C appearing in Figure 6.5. As a result, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$.

The correctness of procedure IsolateMinorC is established by Lemma 9.11. The run-time analysis for procedure IsolateMinorC appears in Lemma 9.12.

Lemma 9.11 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to C, the IsolateMinorC procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

Proof. The proof is the same as in Lemma 9.7 except as follows. Firstly, according to Lemma 9.5, the x-y path in $\tilde{G}$ represented in minor C by the edge (y, z) was previously marked for isolation by the procedure ChooseTypeOfNonplanarityMinor. Secondly, since r' is a root copy of v, the invocation of MarkDFSPath only marks in $\tilde{G}$ the path corresponding to the edge (u, v) in minor C. Thirdly, also because r' is a root copy

of v, the invocation of MarkPathAlongBicompExtFace operates on the edge records and vertex structures representing the cycle $(v, \ldots, x, \ldots, w, \ldots, y, \ldots, v)$ in minor C. Fourthly, the external face path in $\tilde{G}$ corresponding to the edge (v, y) in minor C is not required to form the Kuratowski subgraph. Furthermore, the interpretation of edge (v, y) in minor C is dependent on the input graph G, and more specifically on the point of attachment p_y of the x-y path. If p_y is high (i.e. attached between v and y), then only the path between v and p_y is omitted (except of course the endpoints), rather than omitting the full path along the external face between v and y is omitted (except the endpoints). Note, however, that minor C is representative of a symmetric case in which p_x is not a high point of attachment for the x-y path (as is depicted by vertex z in Figure 6.5). In this case, p_y must be a high point of attachment since, according to Lemma 9.5, the procedure ChooseTypeOfNonplanarityMinor identified minor C due to a high point of attachment for the x-y path. This case is symmetric with the case of having a high point of attachment p_x and low point of attachment p_y, which can be seen by swapping the labels for x and y then flipping the minor C depicted in Figure 6.5. When this case occurs, the procedure exploits the fact that inverting the graph $\tilde{G}$ to match minor C is not necessary because the procedure operates correctly in the symmetric case except for the need to omit the external face path between v and x (rather than v and y). Moreover, there is no need to perform an action symmetric with Line 2 since this symmetric case occurs only when p_x is attached low. Thus, a correspondence has been made with the portion of minor C necessary to mark in $\tilde{G}$ a subgraph of G homeomorphic to $K_{3,3}$. □

Lemma 9.12 *The IsolateMinorC procedure runs in $O(n)$ time.*

Proof. Same arguments as in Lemma 9.8, except for a few additional constant time operations in Lines 1 to 4. □

9.10 Isolating Minor D

The procedure IsolateMinorD defined in Figure 9.7 processes the embedding structure $\tilde{G}$ according to non-planarity minor D appearing in Figure 6.6. As a result,

Figure 9.7: Isolating Minor D

Procedure: IsolateMinorD
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of D
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) MarkPathAlongBicompExtFace(x, y)
(2) MarkDFSPath(min(u_x, u_y), r')
(3) MarkDFSPathsToDescendants()
(4) JoinBicomps()
(5) AddAndMarkUnembeddedEdges()

$\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$.

The correctness of procedure IsolateMinorD is established by Lemma 9.13. The run-time analysis for procedure IsolateMinorD appears in Lemma 9.14.

Lemma 9.13 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to D, the IsolateMinorD procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

Proof. The proof is the same as in Lemma 9.7 except as follows. Firstly, according to Lemma 9.5, the paths in $\tilde{G}$ corresponding to the edges (x, y) and (v, z) in minor D were previously marked for isolation by the procedure ChooseTypeOfNonplanarityMinor. Secondly, since r' is a root copy of v, the invocation of MarkDFSPath only marks in $\tilde{G}$ the path corresponding to the edge (u, v) in minor D. Thirdly, though the vertices v, x and y in minor D are required to form a $K_{3,3}$, the edges (v, x) and (v, y) in minor D are not required. Thus, the parameters to MarkPathAlongBicompExtFace are adjusted so that the vertices and edges are not marked along the corresponding external face paths in $\tilde{G}$ from v to x and v to y (though x and y are marked by MarkPathAlongBicompExtFace, and v is marked in Line 2). Finally, since the procedure ChooseTypeOfNonplanarityMinor only selects minor D if the test for minor C fails, the points of attachment for the x-y path must be low and therefore connect to the portion of the external face marked by MarkPathAlongBicompExtFace. Thus, a correspondence has been made with the portion of minor D necessary to mark in $\tilde{G}$

a subgraph of G homeomorphic to $K_{3,3}$. □

Lemma 9.14 *The IsolateMinorD procedure runs in $O(n)$ time.*

Proof. Same arguments as in Lemma 9.8. □

9.11 Isolating Minor E

Isolating a Kuratowski subgraph from minor E is more challenging than it is for the other minors because minor E is a K_5 minor, which can result in either a $K_{3,3}$ or K_5 homeomorph. The procedure IsolateMinorE, which appears at the end of this section, performs various tests to determine which of five more isolators to invoke. These additional isolators are discussed in the subsections below. They exploit additional structure in the input graph G and embedding structure $\tilde{G}$ that is edge contracted into the vertices of minor E. Four of the isolators are $K_{3,3}$ homeomorph isolators, and the last is a K_5 homeomorph isolator.

9.11.1 Isolating Minor E_1

The procedure IsolateMinorE_1 defined in Figure 9.9 processes the embedding structure $\tilde{G}$ according to the non-planarity minor appearing in Figure 9.8, which is derived from non-planarity minor E appearing in Figure 6.7 by assuming that the externally active vertex along the external face path $(p_x, \ldots, w, \ldots p_y)$ is some vertex z other than the w candidate. As a result of processing by this procedure, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$. The processing consists of a straightforward reduction to minor C, as depicted in Figure 9.8.

The correctness of procedure IsolateMinorE_1 is established by Lemma 9.15. The run-time analysis for procedure IsolateMinorE_1 appears in Lemma 9.16.

Lemma 9.15 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to E and $w \neq z$, the IsolateMinorE_1 procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

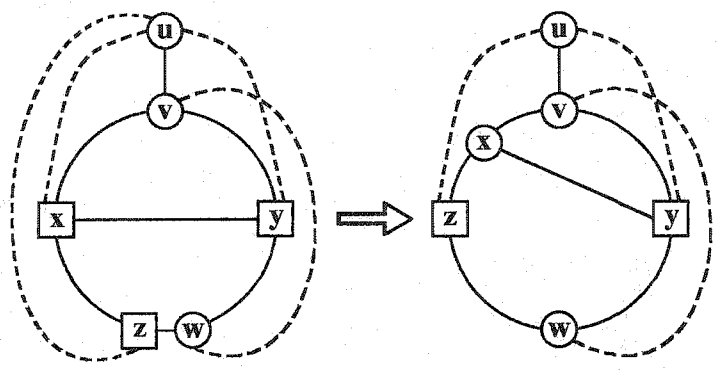


Figure 9.8: Minor E_1 Reduces to Minor C

Figure 9.9: Isolating Minor E_1

Procedure: IsolateMinorE_1
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of E and $w \neq z$.
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) if the type of z is 'low RXW'
(2) Set type along RXW path $(p_x, \ldots, z)$, excluding z, to 'high RXW'
(3) $(x, u_x, d_x) \leftarrow (z, u_z, d_z)$
(4) else Set type along RYW path $(p_y, \ldots, z)$, excluding z, to 'high RYW'
(5) $(y, u_y, d_y) \leftarrow (z, u_z, d_z)$
(6) $(z, u_z, d_z) \leftarrow (nil, nil, nil)$
(7) minorType $\leftarrow$ 'C'
(8) IsolateMinorC()

Proof. Due to the applicability of minor E to $\tilde{G}$, the input requirements of procedure IsolateMinorC are met except that the x-y path does not have a high point of attachment. However, since z is externally active, it meets the requirement of the procedure IsolateMinorC on stopping vertices x and y. Moreover, z appears along the external face path $(p_x, \ldots, w, \ldots, p_y)$, excluding p_x and p_y. If z appears on the RXW path, then it can be substituted for x in minor C if the RXW path vertex types are changed such that p_x becomes a high point of attachment. Otherwise, z appears on the RYW path and can be substituted for y in minor C if the RYW path vertex types are changed such that p_y becomes a high point of attachment. Since the x-y

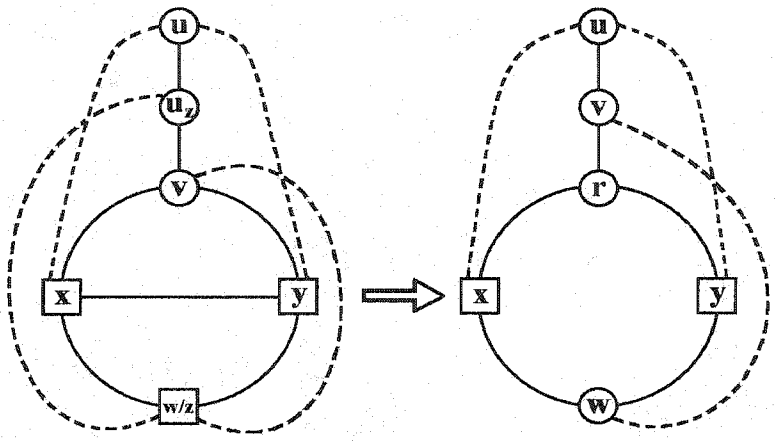


Figure 9.10: Minor E_2 Reduces to Minor A

path has been changed to a high x-y path by substituting z for a stopping vertex, changing the minorType to C is sufficient according to Lemma 9.11 to guarantee that the procedure IsolateMinorC marks a $K_{3,3}$ homeomorph in $\tilde{G}$. □

Lemma 9.16 *The IsolateMinorE_1 procedure runs in $O(n)$ time.*

Proof. All operations are constant time except the $O(n)$ path traversal loops in Lines 2 and 4 and the invocation of IsolateMinorC, which is $O(n)$ by Lemma 9.12. □

9.11.2 Isolating Minor E_2

The procedure IsolateMinorE_2 defined in Figure 9.11 processes the embedding structure $\tilde{G}$ according to the non-planarity minor appearing in Figure 9.10, which is derived from non-planarity minor E appearing in Figure 6.7 as follows. We assume that the w candidate is also the externally active vertex z and that u_z is a descendant of both u_x and u_y. As a result of processing by this procedure, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$. The processing consists of a straightforward reduction to minor A.

The correctness of procedure IsolateMinorE_2 is established by Lemma 9.17. The run-time analysis for procedure IsolateMinorE_2 appears in Lemma 9.18.

Figure 9.11: Isolating Minor E_2

Procedure: IsolateMinorE_2
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of E,
w equal to z, and u_z a descendant of u_x and u_y
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) Assign 0 to 'visited' members of all vertex structures and edge records
(2) Set v in $\tilde{G}$ equal to u_z
(3) $d_w \leftarrow d_z$
(4) $(z, u_z, d_z) \leftarrow (nil, nil, nil)$
(5) minorType $\leftarrow$ 'A'
(6) IsolateMinorA()

Lemma 9.17 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to E, $w = z$ and u_z a descendant of u_x and u_y, the IsolateMinorE_2 procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

Proof. Due to the applicability of minor E to $\tilde{G}$, the input requirements of procedure IsolateMinorA are met except minor A has no x-y path and, in minor A, vertex v is an ancestor of r'. In Line 1, the marked x-y path is unmarked. Furthermore, since u_z is a descendant of u_x and u_y, and since w connects to u_z by a back edge to u_z plus zero or more tree edges from d_z to w, it is possible to substitute u_z for v as long as the edge from u_z to d_z is appropriately represented by assigning d_z to d_w. Note that the external activity status of x and y is unchanged by this substitution because u_x and u_y are ancestors of u_z. Thus, changing the minorType to A is sufficient according to Lemma 9.7 to guarantee that the procedure IsolateMinorA marks a $K_{3,3}$ homeomorph in $\tilde{G}$. □

Lemma 9.18 *The IsolateMinorE_2 procedure runs in $O(n)$ time.*

Proof. All operations are constant time except Line 1, which contains an implied $O(n)$ loop, and the invocation of IsolateMinorA, which is $O(n)$ by Lemma 9.8. □

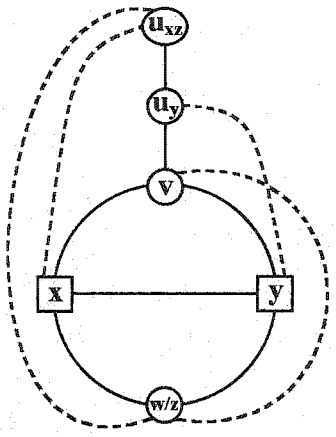


Figure 9.12: Minor E_3 (with u_y a descendant of u_x and u_z)

9.11.3 Isolating Minor E_3

The procedure IsolateMinorE_3 defined in Figure 9.13 processes the embedding structure $\tilde{G}$ according to the non-planarity minor appearing in Figure 9.12, which is derived from non-planarity minor E appearing in Figure 6.7 as follows. We assume that the w candidate is also the externally active vertex z, that u_z is an ancestor of at least one of u_x and u_y, and that u_x does not equal u_y. As a result of processing by this procedure, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$.

Note that minor E_3 as depicted in Figure 9.12 does not require the edges (x, w) and (y, v) to form a $K_{3,3}$. Moreover, while Figure 9.12 depicts u_x as an ancestor of u_y, there is a symmetric case in which u_y is an ancestor of u_x. Finally, the pseudo-code must account for the fact that the points of attachment p_x and p_y of the x-y path have been edge contracted into x and y in minor E_3.

The correctness of procedure IsolateMinorE_3 is established by Lemma 9.19. The run-time analysis for procedure IsolateMinorE_3 appears in Lemma 9.20.

Lemma 9.19 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to E, $w = z$, $u_x \neq u_y$, and u_z an ancestor of at least one of u_x and u_y, the IsolateMinorE_3 procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

Proof. With respect to marking the edges and vertices along paths in $\tilde{G}$ that correspond to the edges (u_{xz}, x), (u_y, y), and (v, w) in minor E_3, the proof is the same

Figure 9.13: Isolating Minor E_3

Procedure: IsolateMinorE_3
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of E,
$w = z$, $u_x \neq u_y$, and u_z an ancestor of at least one of u_x and u_y
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) if $u_x < u_y$
(2) MarkPathAlongBicompExtFace(r', p_x)
(3) MarkPathAlongBicompExtFace(w, y)
(4) else MarkPathAlongBicompExtFace(x, w)
(5) MarkPathAlongBicompExtFace(p_y, r')

(6) MarkDFSPath(min(u_x, u_y, u_z), r')
(7) MarkDFSPathsToDescendants()
(8) JoinBicomps()
(9) AddAndMarkUnembeddedEdges()

as for the edges (u, x), (u, y), and (v, w) in Lemma 9.7. Likewise, the path in $\tilde{G}$ corresponding to the edge (u_{xz}, w) is isolated by exploiting the same methods as those used to mark the path in $\tilde{G}$ corresponding to the path $(u_{xz}, \ldots, d_x, \ldots, x)$ in minor E_3. According to Lemma 9.5, the x-y path in $\tilde{G}$ represented in minor E_3 by the edge (x, y) was previously marked for isolation by the procedure ChooseTypeOfNonplanarityMinor.

There are a number of variations on the depiction of minor E_3 in Figure 9.12. A symmetric case occurs when u_x is the descendant of u_y and u_z (which, for the purpose of depiction, would be contracted to u_{yz}). Moreover, u_x and u_z (or u_y and u_z in the symmetric case) need not be equivalent. With respect to marking the edges and vertices along paths in $\tilde{G}$ that correspond to edges (u_{xz}, u_y) and (u_y, v) depicted in Figure 9.12, all variations and symmetric cases are accounted for by simply marking the DFS path from v up to the minimum of the ancestors u_x, u_y, and u_z, which occurs in Line 6.

All of the vertices and edges corresponding to minor E_3 have been discussed except for the cycle $(v, \ldots, x, \ldots, w, \ldots, y, \ldots, v)$, which corresponds in $\tilde{G}$ to the external face cycle of the biconnected component B rooted by r' (except that v, the

parent copy of r', is marked when the edge (v, d_w) is added). When u_x is the ancestor of u_y, the portions of the input graph corresponding to the edges (x, w) and (y, v) in Figure 9.12 are not required to form the $K_{3,3}$ homeomorph. In the symmetric case in which u_y is the ancestor of u_x, the portions of the input graph corresponding to edges (y, w) and (x, v) are not required. The procedure accounts for these two cases in Lines 1 to 5. Moreover, the procedure accounts for the fact that the points of attachment p_x and p_y of the x-y path are edge contracted into x and y in minor E_3. When marking the path in $\tilde{G}$ corresponding to edge (v, x), it extends the marking to p_x in Line 2 in order to connect to the x-y path. Similarly, if p_y is between w and y such that p_y becomes one of the degree three vertices in the $K_{3,3}$ homeomorph, Line 3 still marks the path in $\tilde{G}$ from w to y since the connection must be made from p_y to u_y. Symmetric arguments hold when u_y is the ancestor of u_x. The invocation of JoinBicomps in Line 8 completes the correspondence with the portion of minor E_3 necessary to mark in $\tilde{G}$ a subgraph of G homeomorphic to $K_{3,3}$. □

Lemma 9.20 *The IsolateMinorE_3 procedure runs in $O(n)$ time.*

Proof. Same arguments as in Lemma 9.8, except for a few extra constant time operations. □

9.11.4 Isolating Minor E_4

The procedure IsolateMinorE_4 defined in Figure 9.15 processes the embedding structure $\tilde{G}$ according to the non-planarity minor appearing in Figure 9.14, which is derived from non-planarity minor E appearing in Figure 6.7 as follows. We assume that the w candidate is also the externally active vertex z and that two of u_x, u_y and u_z are equal and not ancestors of the third. Moreover, given the points of attachment p_x and p_y of the x-y path, we assume that at least one of the points of attachment is not equal to the corresponding stopping vertex (i.e., at least one of $p_x \neq x$ or $p_y \neq y$ holds). As a result of processing by this procedure, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$.

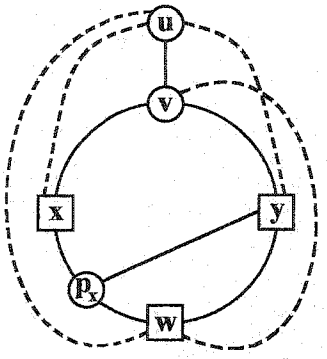


Figure 9.14: Minor E_4 with p_x not equal to x

Figure 9.15: Isolating Minor E_4

Procedure: IsolateMinorE_4
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of E, $w = z$, at least two of u_x, u_y and u_z are equal and not ancestors of the third, and $p_x \neq x$ or $p_y \neq y$ (or both).
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) if $p_x \neq x$
(2) MarkPathAlongBicompExtFace(r', w)
(3) MarkPathAlongBicompExtFace(p_y, r')
(4) else MarkPathAlongBicompExtFace(r', p_x)
(5) MarkPathAlongBicompExtFace(w, r')

(6) MarkDFSPath(min(u_x, u_y, u_z), max(u_x, u_y, u_z))
(7) MarkDFSPathsToDescendants()
(8) JoinBicomps()
(9) AddAndMarkUnembeddedEdges()

Note that minor E_4 as depicted in Figure 9.14 does not require the edges (u, v) and (w, y) to form a $K_{3,3}$. Moreover, while Figure 9.14 depicts p_x distinct from x (and p_y can be equal to or distinct from y), there is a symmetric case in which p_x is equal to x and therefore p_y must be distinct from y.

The correctness of procedure IsolateMinorE_4 is established by Lemma 9.21. The run-time analysis for procedure IsolateMinorE_4 appears in Lemma 9.22.

Lemma 9.21 *Given the embedding structure $\tilde{G}$ in the isolator initial state with mi-*

norType equal to E, $w = z$, at least two of u_x, u_y and u_z are equal and not ancestors of the third, and $p_x \neq x$ or $p_y \neq y$ (or both), the IsolateMinorE$_4$ procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.

Proof. With respect to marking the edges and vertices along paths in $\tilde{G}$ that correspond to the edges (u, x), (u, y), and (v, w) in minor E_4, the proof is the same as in Lemma 9.7. Likewise, the path in $\tilde{G}$ corresponding to the edge (u, w) is isolated by the same methods as those used to isolate the path $(u_x, \ldots, d_x, \ldots, x)$. The path in $\tilde{G}$ represented by vertex u is marked by the invocation of MarkDFSPath in Line 6. Note that the path in $\tilde{G}$ corresponding to the edge (u, v), excluding endpoints, is not marked because the edge is not needed to form a $K_{3,3}$ in minor E_4 (except the endpoints, which are marked by other operations in Lines 7 and 9). As for the x-y path in $\tilde{G}$ represented in minor E_4 by the edge (p_x, y), Lemma 9.5 stipulates that it was previously marked for isolation by the procedure ChooseTypeOfNonplanarityMinor.

All of the vertices and edges corresponding to minor E_4 have been discussed except for the cycle $(v, \ldots, x, \ldots, w, \ldots, y, \ldots, v)$, which corresponds in $\tilde{G}$ to the external face cycle of the biconnected component B rooted by r' (except that v, the parent copy of r', is marked when the edge (v, d_w) is added). If p_x is not equal to x, then the edge (w, y) in minor E_4 is not required to form a $K_{3,3}$, so Lines 2 and 3 mark the external face of B except the path from w to p_y (though the endpoints are marked later). If p_x equals x, but p_y does not equal y, then symmetric operations are applied in Lines 4 and 5. The invocation of JoinBicomps in Line 8 completes the correspondence with the portion of minor E_4 necessary to mark in $\tilde{G}$ a subgraph of G homeomorphic to $K_{3,3}$. □

Lemma 9.22 *The IsolateMinorE$_4$ procedure runs in $O(n)$ time.*

Proof. Same arguments as in Lemma 9.8, except for a few extra constant time operations. □

Figure 9.16: Isolating Minor E_5

Procedure: IsolateMinorE_5
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of E,
$w = z$, $p_x = x$, $p_y = y$, and at least two of u_x, u_y and u_z are
equal and not ancestors of the third.
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) MarkPathAlongBicompExtFace(r', r')
(2) MarkDFSPath(min(u_x, u_y, u_z), r')
(3) MarkDFSPathsToDescendants()
(4) JoinBicomps()
(5) AddAndMarkUnembeddedEdges()

9.11.5 Isolating Minor E_5

The procedure IsolateMinorE_5 defined in Figure 9.16 processes the embedding structure $\tilde{G}$ according to the non-planarity minor E appearing in Figure 6.7. We assume that the w candidate is also the externally active vertex z, that at least two of u_x, u_y and u_z are equal and not ancestors of the third, and that the points of attachment p_x and p_y of the x-y path are equal to the corresponding stopping vertices (i.e., $p_x = x$ and $p_y = y$). As a result of processing by this procedure, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to K_5.

The correctness of procedure IsolateMinorE_5 is established by Lemma 9.23. The run-time analysis for procedure IsolateMinorE_5 appears in Lemma 9.24.

Lemma 9.23 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to E, $w = z$, $p_x = x$, $p_y = y$, and at least two of u_x, u_y and u_z are equal and not ancestors of the third, the IsolateMinorE_5 procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

Proof. With respect to marking the edges and vertices along paths in $\tilde{G}$ that correspond to the edges (u, x), (u, y), and (v, w) in minor E, the proof is the same as in Lemma 9.7. The paths in $\tilde{G}$ represented by vertex u and the edge (u, v) are marked by a single invocation of MarkDFSPath in Line 2. The path in $\tilde{G}$ corresponding to

the edge (u, w) is isolated by the same methods as those used to isolate the path (u_x, ..., d_x, ..., x). Since r' is a root copy of v, the invocation of MarkPathAlongBicompExtFace marks the cycle in $\tilde{G}$ corresponding to the cycle (v, ..., x, ..., w, ..., y, ..., v) in minor E (except that v, the parent copy of r', is marked when the edge (v, d_w) is added). Finally, according to Lemma 9.5, the x-y path in $\tilde{G}$ represented in minor E by the edge (x, y) was previously marked for isolation by the procedure ChooseTypeOfNonplanarityMinor. The invocation of JoinBicomps in Line 4 completes the correspondence with minor E such that a subgraph of G homeomorphic to K_5 has been marked in $\tilde{G}$. □

Lemma 9.24 *The IsolateMinorE_5 procedure runs in $O(n)$ time.*

Proof. Same arguments as in Lemma 9.8, except for a few extra constant time operations. □

9.11.6 Closure on Minor E

The procedure IsolateMinorE defined in Figure 9.17 processes the embedding structure $\tilde{G}$ according to non-planarity minor E appearing in Figure 6.7. The procedure performs a number of tests to determine whether the G and $\tilde{G}$ satisfy the input requirements of each of the five isolators defined earlier in this section. As a result of invoking the selected isolator, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$ or K_5.

The correctness of procedure IsolateMinorE is established by Lemma 9.25. The run-time analysis for procedure IsolateMinorE appears in Lemma 9.26.

Lemma 9.25 *Given the embedding structure $\tilde{G}$ in the isolator initial state with minorType equal to E, the IsolateMinorE procedure transforms $\tilde{G}$ to the Kuratowski subgraph marked state.*

Proof. The vertex u in minor E is representative of three possibly distinct vertices u_x, u_y, and u_z. The points of attachment p_x and p_y are edge contracted into x and

Figure 9.17: Isolating Minor E

Procedure: IsolateMinorE
this: Embedding structure $\tilde{G}$ in the isolator initial state with minorType of E
out: Embedding structure $\tilde{G}$ in the Kuratowski subgraph marked state

(1) if $z \neq w$, IsolateMinorE$_1$()
(2) else if $u_z > \max(u_x, u_y)$, IsolateMinorE$_2$()
(3) else if $u_z < \max(u_x, u_y)$ and $u_x \neq u_y$, IsolateMinorE$_3$()
(4) else if $p_x \neq x$ or $p_y \neq y$, IsolateMinorE$_4$()
(5) else IsolateMinorE$_5$()

y and may be distinct from x and y. The vertex w is pertinent, and either it is externally active or their is another externally active vertex on the external face path $(p_x, \ldots, w, \ldots, p_y)$. Let z denote this externally active vertex.

The following cases result from the variability of the identified vertices u_x, u_y, u_z, p_x, p_y, and z. Either $z \neq w$ (minor E$_1$) or $z = w$. In the latter case, we consider the mutually exclusive cases for u_x, u_y, and u_z.

1. u_z is descendant to both u_x and u_y (minor E$_2$).

2. u_z is an ancestor of at least one of u_x and u_y and $u_x \neq u_y$ (minor E$_3$).

3. u_z is an ancestor of $u_x = u_y$ (minor E$_4$ or minor E$_5$).

4. u_z equal to one or both of u_x and u_y (minor E$_4$ or minor E$_5$).

In both of the latter two cases, at least two of u_x, u_y, and u_z are equal to and not an ancestor of the third. When this condition occurs, minors E$_4$ and E$_5$ are distinguished by the fact that either the x-y path is attached directly to x and y (minor E$_5$), or it is not (minor E$_4$).

Since vertices v, x, y and w in minor E are fixed, the cases above account for all variability in minor E. Thus, the fact that $\tilde{G}$ is transformed to the Kuratowski subgraph marked state follows from Lemmas 9.15, 9.17, 9.19, 9.21, and 9.23. □

Figure 9.18: Isolating a Kuratowski Subgraph

Procedure: IsolateKuratowskiSubgraph
this: Embedding structure $\tilde{G}$ in the non-planar embedding invariant state
in: Input graph G
out: Embedding structure $\tilde{G}$ containing a subgraph of G homeomorphic to $K_{3,3}$ or K_5

(1) ChooseTypeOfNonplanarityMinor(G)
(2) InitializeIsolatorContext()

(3) if minorType is 'A', IsolateMinorA()
(4) else if minorType is 'B', IsolateMinorB()
(5) else if minorType is 'C', IsolateMinorC()
(6) else if minorType is 'D', IsolateMinorD()
(7) else if minorType is 'E', IsolateMinorE()

(8) Delete unmarked vertices and edges

Lemma 9.26 *The IsolateMinorE procedure runs in $O(n)$ time.*

Proof. In addition to a few constant time operations, this procedure invokes one of IsolateMinorE$_1$, IsolateMinorE$_2$, IsolateMinorE$_3$, IsolateMinorE$_4$, and IsolateMinorE$_5$, which are each $O(n)$ by Lemmas 9.16, 9.18, 9.20, 9.22, and 9.24. □

9.12 Isolating a Kuratowski Subgraph

The procedure IsolateKuratowskiSubgraph defined in Figure 9.18 receives the input graph G and the embedding structure $\tilde{G}$ in the non-planar embedding invariant state (Definition 9.1), which is the defined result of the procedure PlanarityTest in Chapter 7 on a non-planar input graph. As a result of processing by this procedure, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$ or K_5.

The correctness of procedure IsolateKuratowskiSubgraph is established by Theorem 9.27. The run-time analysis for procedure IsolateKuratowskiSubgraph appears in Theorem 9.28.

Theorem 9.27 *Given the embedding structure $\tilde{G}$ in the non-planar embedding invariant state, the IsolateKuratowskiSubgraph procedure modifies $\tilde{G}$ such that it contains a subgraph of the input graph G that is homeomorphic to $K_{3,3}$ or K_5.*

Proof. Lines 1 and 2 transform $\tilde{G}$ to the isolator initial state. According to Lemma 9.5, the test-and-branch decision construct in Lines 3 to 7 correctly chooses to invoke one of the procedures IsolateMinorA, IsolateMinorB, IsolateMinorC, IsolateMinorD, and IsolateMinorE, which mark in $\tilde{G}$ a subgraph of G homeomorphic to $K_{3,3}$ or K_5 according to Lemmas 9.7, 9.9, 9.11, 9.13 and 9.25. The procedure completes the isolation of a Kuratowski subgraph by simply removing the unmarked vertices and edges. □

Theorem 9.28 *The IsolateKuratowskiSubgraph procedure runs in $O(n)$ time.*

Proof. Line 1 invokes ChooseTypeOfNonplanarityMinor, which is $O(n)$ by Lemma 9.6. Line 2 invokes a simple, low-level procedure easily seen to be $O(n)$. Lines 3 to 7 contain constant time operations except for the procedure invocations of IsolateMinorA, IsolateMinorB, IsolateMinorC, IsolateMinorD, and IsolateMinorE, which are $O(n)$ by Lemmas 9.8, 9.10, 9.12, 9.14 and 9.26. Line 8 is clearly $O(n)$. □

10. New Outerplanar Graph Testing, Embedding and Obstruction Isolation

This chapter extends the theoretical framework for planarity created in the prior chapters of this dissertation to solve the testing, embedding and minimal obstructing subgraph problems for outerplanarity. Section 10.1 is devoted to outerplanarity testing and embedding. Section 10.2 exploits the proof of correctness to obtain a subgraph homeomorphic to $K_{2,3}$ or K_4 when it is found that a given graph is not outerplanar. A prior linear time algorithm for outerplanar graph embedding is due to Brehaut [9], which is based on the Hopcroft and Tarjan planarity test [30].

10.1 Outerplanarity Testing and Embedding

In the planarity algorithms, the notion of external activity serves to indicate vertices that must remain on the external face. Since an outerplanar graph requires all vertices to be on external face, our planar graph testing and embedding algorithms can be transformed to testing and embedding algorithms for outerplanarity by simply changing the definition of external activity to be a permanent state of the parent copy of every vertex. From an implementation standpoint, this change could be accomplished by changing the VertexActiveStatus procedure defined in Chapter 4 to simply return 'Externally Active' at all times.

Property 10.1 *For outerplanarity algorithms, all vertices are permanently externally active.*

Definition 10.1 *The embedding is in the* outerplanar embedding invariant state *if it conforms to the embedding invariant state of Definition 3.8 except for maintaining Property 10.1 in lieu of Property 3.6 (Dynamic External Activity).*

Given that all vertices are defined to be externally active at all times, if the procedure PlanarityTest defined in Chapter 7 is able to embed all back edges between

each vertex and its descendants, then clearly the input graph is outerplanar, and an embedding can be recovered by invoking the procedure PlanarEmbed defined in Chapter 8. If, however, the underlying procedure Walkdown defined in Chapter 6 fails to embed a back edge during a step v, then the input graph G is not outerplanar, as shown by Theorem 10.1.

Theorem 10.1 *Given an embedding structure $\tilde{G}$ in the outerplanar embedding invariant state (Definition 10.1) and a root copy v' in $\tilde{G}$ with DFS child c, the actions of the Walkdown procedure result in an embedding structure $\tilde{G}$ in the outerplanar embedding invariant state. If the graph G is outerplanar or if neither Condition 1 or 2 occur, then $\tilde{G}$ contains all back edges from the input graph G between v and descendants of c.*

Proof. If the Walkdown embeds all back edges between v^c and descendants of c, then the resulting partial embedding remains outerplanar due to maintenance of Properties 10.1 (all vertices are externally active) and 3.2 (active vertices are maintained on the external face). Thus, we focus on the conditions under which the Walkdown fails to embed one or more back edges between v^c and descendants of c. Note that it is well-known (e.g., see [59]) that outerplanarity is obstructed by homeomorphs and hence minors of $K_{2,3}$ and K_4.

If the Walkdown returns with a non-empty stack (Condition 1), then edge contraction and deletion can be used on the embedding structure $\tilde{G}$, along with the addition of certain unembedded edges represented in $\tilde{G}$ as vertex activity, to produce the non-outerplanarity minor A appearing in Figure 10.1. Thus, the input graph is not outerplanar since Figure 10.1 is a $K_{2,3}$.

If the Walkdown returns with an empty stack (Condition 2) and there is a w candidate with a pertinent child biconnected component, then edge contraction and deletion can be used on the embedding structure $\tilde{G}$, along with the addition of certain unembedded edges represented in $\tilde{G}$ as vertex activity, to produce the non-outerplanarity minor B appearing in Figure 10.2. Thus, the input graph is not outerplanar since Figure 10.2 is a $K_{2,3}$.

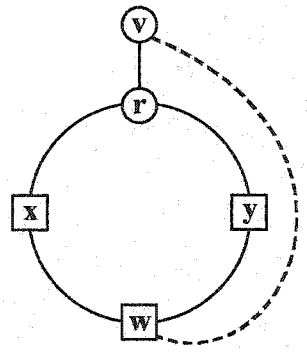


Figure 10.1: Non-Outerplanarity Minor A

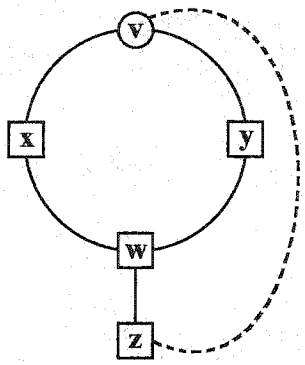


Figure 10.2: Non-Outerplanarity Minor B

If the Walkdown terminates on Condition 2, and there is no x-y path for a w candidate, then the Walkdown visits w prior to merging w, x and y into the same biconnected component. Thus, a w candidate must have a pertinent child biconnected component if it has no x-y path.

If the Walkdown returns with an empty stack (Condition 2) and the biconnected component rooted by v' has an x-y path, then edge contraction and deletion can be used on the embedding structure $\tilde{G}$, along with the addition of certain unembedded edges represented in $\tilde{G}$ as vertex activity, to produce the non-outerplanarity minor E appearing in Figure 10.3. Thus, the input graph is not outerplanar since Figure 10.3 is a K_4.

Thus, when the Walkdown fails to embed a back edge, the input graph is not outerplanar. If the halt occurred when the stack was not empty, then the input graph contains a $K_{2,3}$ minor. If the stack is empty, then a w candidate, which has an unembedded back edge to v at the end of step v, either has or does not have a

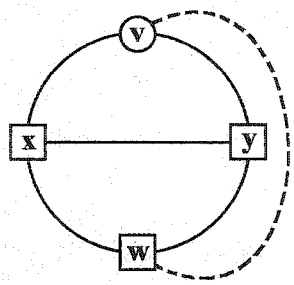


Figure 10.3: Non-Outerplanarity Minor E

blocking x-y path. If there is a blocking x-y path, then the input graph contains a K_4 minor and is therefore not outerplanar. If there is no x-y path, then the w candidate must have a pertinent child biconnected component, so the input graph contains a $K_{2,3}$ minor and is therefore not outerplanar. □

Theorem 10.2 *A simple undirected graph G with n vertices, an outerplanar embedding of G can be created or shown not to exist in $O(n)$ time.*

Proof. Follows directly from Theorems 7.2 and 10.1. □

It is interesting to note that non-planarity minors C and D have no outerplanarity analogs. Non-planarity minor C requires the existence of an x-y path with at least one high point of attachment. For outerplanarity, the algorithm makes all vertices externally active, so the stopping vertices x and y are endpoints of the two edges incident to the biconnected component root along the external face. Therefore, it is impossible to have a high x-y path. An analog for non-planarity minor D does not exist because it contains an internal vertex z, which cannot occur with our algorithm since all vertices are externally active.

10.2 Isolating a Minimal Subgraph that Obstructs Outerplanarity

The procedure IsolateOuterplanarityObstruction defined in Figure 10.4 receives the input graph G and the embedding structure $\tilde{G}$ in the outerplanar embedding

Figure 10.4: Outerplanarity Obstruction Isolator

```
Procedure: IsolateOuterplanarityObstruction
this: Embedding structure G̃ in the outerplanar embedding invariant state
in: The input graph G
out: Embedding structure G̃ in the non-planarity minor found state

(1) OrientVertices()
(2) InitializeNonplanarityMinorContext(G)

(3) if r' is not a root copy of v
(4)    Assign 'A' to minorType
(5) else if the pertinentBicompList of w is not empty
(6)       Assign 'B' to minorType
(7) else Assign 'E' to minorType

(8) Obtain d_w in the manner described in Table 9.2

(9) if minorType is 'E', MarkHighestXYPath()

(10) MarkPathAlongBicompExtFace(r', r')
(11) MarkDFSPath(v, r')
(12) MarkDFSPath(w, d_w)
(13) JoinBicomps()
(14) Add and mark the edge (v, d_w)

(15) Delete unmarked edges and vertices
```

invariant state. Moreover, the Walkdown is assumed to have experienced a Condition 1 or 2 termination during some step v. As a result of processing by this procedure, $\tilde{G}$ contains a subgraph of the input graph G that is homeomorphic to $K_{2,3}$ or K_4.

The correctness of procedure IsolateOuterplanarityObstruction is established by Theorem 10.3. The run-time analysis for procedure IsolateOuterplanarityObstruction appears in Theorem 10.4.

Theorem 10.3 *Given the embedding structure $\tilde{G}$ in the outerplanar embedding invariant state on which the Walkdown has produced a Condition 1 or 2 termination, the IsolateOuterplanarityObstruction procedure modifies $\tilde{G}$ such that it contains a subgraph of the input graph G that is homeomorphic to $K_{2,3}$ or K_4.*

Proof. Much of the proof follows from Theorem 10.1. The edge (x, y) in minor E is marked in Line 9. Line 10 marks the cycle $(r, \ldots, x, \ldots, w, \ldots, y, \ldots, r)$ in minor A, or the cycle $(v, \ldots, x, \ldots, w, \ldots, y, \ldots, v)$ for minors B and E. Line 11 has no meaningful effect except for minor A, where it marks the tree path corresponding to the edge (v, r). The combined action of Lines 12 and 14 handle the path in $\tilde{G}$ corresponding to (v, w) in minors A and E, or the path $(v, \ldots, z, \ldots, w)$. Since the biconnected components are also merged, eliminating root copies, the correspondence with the minors depicted in Figures 10.1, 10.2 and 10.3 is complete, so the procedure completes the isolation of an outerplanarity obstruction by simply removing the unmarked vertices and edges. □

Theorem 10.4 *The IsolateOuterplanarityObstruction procedure runs in $O(n)$ time.*

Proof. Line 1 is $O(n)$ by Lemma 8.2. Line 2 invokes a low-level procedure containing simple $O(n)$ loops. Lines 3 to 8 are constant time. Line 9 contains a constant time comparison plus an invocation of MarkHighestXYPath, which is $O(n)$ by Lemma 9.2. The remaining operations invokes low-level procedures containing simple $O(n)$ loops, except Line 13, which is $O(n)$ by Lemma 4.21, and Line 14 which requires constant time. □

11. New $K_{2,3}$ and $K_{3,3}$ Search Algorithms

This chapter extends the theoretical framework for planarity and outerplanarity created in the prior chapters of this dissertation to solve the problems of searching for $K_{2,3}$ and $K_{3,3}$ homeomorphs in a graph, and recognition of $K_{2,3}$-less and $K_{3,3}$-less graphs (i.e. graphs having no $K_{2,3}$ or $K_{3,3}$ homeomorph, respectively). Section 11.1 presents a simple $O(n)$ search algorithm for $K_{2,3}$ homeomorphs, and Section 11.2 presents a simple $O(n)$ search algorithm for $K_{3,3}$ homeomorphs. Prior linear time algorithms for these problems are due to Asano [4] and Fellows and Kaschube [20].

11.1 Searching for $K_{2,3}$ Homeomorphs and Recognizing $K_{2,3}$-less Graphs

The $K_{2,3}$ homeomorph search algorithm is essentially the outerplanarity algorithm of Chapter 10, except that further analysis of the graph is performed if non-outerplanarity minor E (the K_4 obstruction) is found. An algorithm nearly identical to the Walkup of Chapter 5 can be used to expediently identify each root copy v^c of the current vertex v upon which the Walkdown terminated with Condition 1 or 2. To process a root copy v^c on which a Condition 1 or 2 Walkdown failure has occurred, we begin by preprocessing only the vertices and edges in the biconnected component rooted by v^c, including vertex orientation, clearing of the visited members, finding x, y and w, and classifying the external face vertices as 'high RXW', 'low RXW', 'high RYW' or 'low RYW' (short-circuit edge removal is necessary because none are added). Then, we determine whether the non-outerplanarity minor is A, B or E (see Lines 3 to 7 of procedure IsolateOuterplanarityObstruction in Figure 10.4). If minors A or B are found, then the desired $K_{2,3}$ homeomorph is isolated and returned to the caller. If minor E is found, then the x-y path is obtained, and the additional analyses described below in Theorem 11.1 are performed, resulting either in the isolation of a $K_{2,3}$ homeomorph or a determination that the K_4 can be ignored. If, for each root copy v^c, the determination is made that the K_4 can be ignored, then the outerpla-

narity algorithm can simply continue at step $v-1$ (except $v=0$) as if an obstruction to outerplanarity had not occurred. If step $v=0$ is completed without finding any $K_{2,3}$ homeomorphs, then the input graph is $K_{2,3}$-less.

Theorem 11.1 *In the outerplanarity testing algorithm, if the Walkdown fails to embed a back edge to a descendant of c of the biconnected component with root v^c, then either a $K_{2,3}$ homeomorph can be found in the input graph G, or the vertices in the DFS subtree rooted by c are determined not to be part of a $K_{2,3}$ homeomorph of G.*

Proof. Non-outerplanarity minors A and B (Figures 10.1 and 10.2) indicate $K_{2,3}$ homeomorphs, so further analysis pertains to non-outerplanarity minor E (Figure 10.3). If either point of attachment of the x-y path is attached below x or y (i.e. if $p_x \neq x$ or $p_y \neq y$), then a $K_{2,3}$ homeomorph can by isolated according to Figure 11.1(a) (note the simple variation cases of having $p_x = z \neq x$, $p_y = z \neq y$, and having both $p_x \neq x$ and $p_y \neq y$).

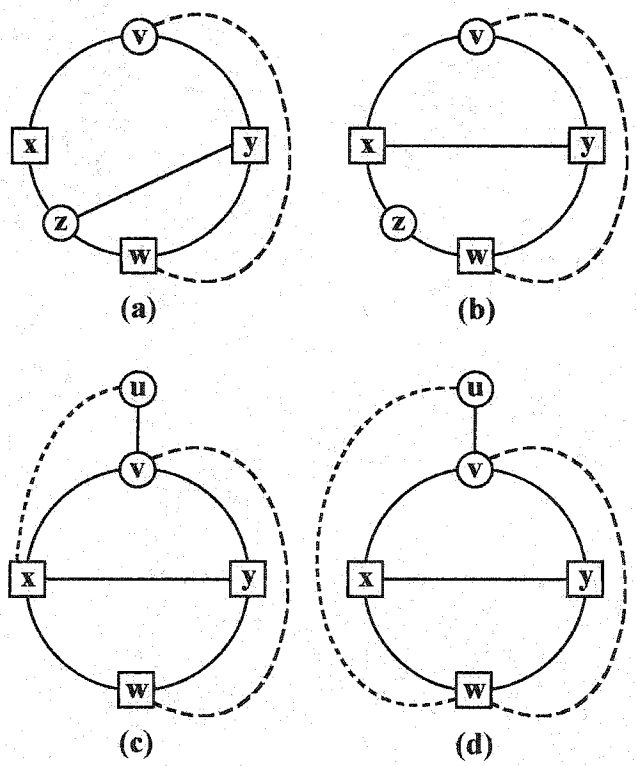


Figure 11.1: $K_{2,3}$ Homeomorphs from Non-Outerplanarity Minor E

Thus, consider the case in which the x-y path (a single edge) is attached directly to x and y. If any other vertex z is on the lower external face path (x, ..., w, ..., y), then a $K_{2,3}$ homeomorph can be isolated according to Figure 11.1(b) (note the symmetric case of having z along the lower external face path between w and y).

Thus, w must be the only vertex along the lower external face path, and we already know that the connection from w to v must be an edge (otherwise, non-outerplanarity minor B). Since we also know that x and y are direct neighbors of each other and of v, the vertices v, x, y and w form a clique. If x or y can be connected to an ancestor u of v by zero or more separated biconnected components plus an unembedded back edge, then a $K_{2,3}$ homeomorph can be isolated according to Figure 11.1(c) (note symmetric case of a connection from y to u). If w can be connected to an ancestor u of v by zero or more separated biconnected components plus an unembedded back edge, then a $K_{2,3}$ homeomorph can be isolated according to Figure 11.1(d).

Finally, if none of the above conditions are met, then every path in G between vertices in the DFS subtree T_c rooted by c (i.e. x, y and w) and the ancestors of v (if any) and descendants of v not in T_c (if any) contains v. Thus, x, y and w are not in any $K_{2,3}$ homeomorph that may be in the input graph. □

Corollary 11.2 *A non-outerplanar biconnected component of a graph is isomorphic to K_4 or contains a $K_{2,3}$ homeomorph.*

Theorem 11.3 *Given a simple undirected graph G with n vertices, a subgraph homeomorphic to $K_{2,3}$ can be found or shown not to exist in G in $O(n)$ time.*

Proof. A $K_{2,3}$-less graph is planar, and a non-planar graph contains a $K_{2,3}$ homeomorph, so the edge limit of Corollary 1.5 is applicable. The cost of identifying root copies of v on which the Walkdown terminated with Condition 1 or 2 is commensurate with the cost of the Walkup operations performed in step v. For a given root v^c of a biconnected component B, the determination of minor A, B or E and the marking of the x-y path for minor E have costs commensurate with the number of vertices in B.

Since B either becomes part of a $K_{2,3}$ homeomorph or is a K_4 that is never visited again, the $O(n)$ total run-time is maintained.

The tests corresponding to Figures 11.1(a) and 11.1(b) are clearly constant time. The tests for Figures 11.1(c) and 11.1(d) are constant time using the VertexActiveStatus procedure of Chapter 4 (i.e. x, y or w has a connection to an ancestor u of v if its leastAncestor field is less than v or if the lowpoint of the first node in its separatedDFSChildList is less than v). If one of the tests determines that a $K_{2,3}$ homeomorph exists, then the $K_{2,3}$ homeomorph is isolated in $O(n)$ time (the time to isolate minor A, B or E plus the time to isolate the path from x, y or w to an ancestor of v in Figures 11.1(c) and 11.1(d)). □

11.2 Searching for $K_{3,3}$ Homeomorphs and Recognizing $K_{3,3}$-less Graphs

A *separation pair*, also called a 2-cut, is a pair of vertices u and v in a biconnected component of a graph G that, if removed, would result in a graph with more connected components than G. A biconnected component is *triconnected* if it does not contain a separation pair. Hopcroft and Tarjan [28] and Vo [56] define and describe methods for obtaining the triconnected components of a graph.

The $K_{3,3}$ homeomorph search algorithm is analogous in operation to the $K_{2,3}$ homeomorph search algorithm. Indeed, the parallels are striking provided that we consider the behavior of the $K_{3,3}$ search algorithm over the triconnected components of an input graph, which is appropriate due to Theorem 11.4. The algorithm reported in this section searches for $K_{3,3}$ homeomorphs in linear time when applied to the triconnected components of a graph, but it is also quite fast in practice, achieving linear time without the need for separating triconnected components as long as a given K_5 homeomorph does not share with other K_5 homeomorphs the ancestral path from the vertex v in which the former is discovered.

Theorem 11.4 (Asano [4]) *A graph G has a subgraph homeomorphic to $K_{3,3}$ if and only if a 3-connected component of G has a subgraph homeomorphic to $K_{3,3}$.*

Remark: A triconnected component can contain one or more *virtual edges* to represent portions of the input graph separated from the vertices of the triconnected component by a separation pair. The endpoints of a virtual edge are a separation pair. This is analogous to creating extra copies of vertices to represent cut vertices in all biconnected components that contain them.

The $K_{3,3}$ search algorithm is essentially the planarity algorithm of Chapter 7, except that further analysis of the graph is performed if non-planarity minor E (the K_5 minor obstruction) is found. An algorithm nearly identical to the Walkup of Chapter 5 can be used to expediently identify each root copy v^c of the current vertex v upon which the Walkdown terminated with Condition 1 or 2. To process a root copy v^c, we begin by preprocessing only the vertices and edges in the biconnected component B rooted by v^c, including vertex orientation, short-circuit edge removal, clearing of the visited members, finding x, y and w, and classifying the external face vertices as 'high RXW', 'low RXW', 'high RYW' or 'low RYW'. Then, we determine which non-planarity minor can be identified. If the non-planarity minor identified is not minor E, then the desired $K_{3,3}$ homeomorph is isolated and returned to the caller. If minor E is found, then the x-y path has also been obtained, so the additional analyses described below in Theorem 11.5 are performed, resulting either in the isolation of a $K_{3,3}$ homeomorph or a determination that the K_5 homeomorph can be planarized. If, for each root copy v^c, the determination is made that the K_5 homeomorph can be planarized, then the planarity algorithm can simply continue at step $v-1$ (except $v=0$) as if an obstruction to planarity had not occurred. If step $v=0$ is completed without finding any $K_{3,3}$ homeomorphs, then the input graph is $K_{3,3}$-less.

The following terms and notation aid the presentation of Theorem 11.5. Let F_x denote descendants of a vertex x in the DFS subtrees rooted by children of x that are in the externally active child biconnected components of x. An *external connection* from a vertex x to an ancestor u of the current vertex v is a path between x and u consisting of a single back edge, or a path of tree edges from x to a descendant in F_x plus a back edge from the descendant to u. Let f_w denote descendants of a vertex w in the DFS subtrees rooted by children of w that are in internally active

child biconnected components of w. An *internal connection* from a vertex w to the current vertex v is a path between w and v consisting of a single back edge or a path of tree edges from w to a descendant in f_w plus a back edge from the descendant to v.

Let u_x, u_y and u_w denote the ancestors of v with the least DFI having an external connection to x, y and w, respectively. Let u_{min} and u_{max} denote the minimum and maximum, respectively, of u_x, u_y and u_w. Let T_c denote the DFS subtree with root c (the child of v^c in the root edge of B), and let P denote the path $(v, \ldots, u_{min})$. Consider the bridges of G with respect to path P. The T_c-bridge contains the vertices in T_c (plus the points of connection to P). Let β_P denote the set of all bridges of G with respect to P except the T_c-bridge. A bridge in β_P *straddles* the vertex u_{max} if the bridge attaches to ancestors and descendants of u_{max} in P. Let β_{P+B} denote the set of all bridges of G with respect to the subgraph of G containing the vertices of P and of the biconnected component B. Aside from the bridges of β_P, the set β_{P+B} contains four important bridges: the F_x-bridge containing the vertices of forest F_x (plus x and one or more ancestors of v), the F_y-bridge containing the vertices of forest F_y (plus y and one or more ancestors of v), the F_w-bridge containing the vertices of forest F_w (plus w and one or more ancestors of v), and the f_w-bridge containing the vertices of forest f_w (plus w and v) .

Theorem 11.5 *In the planarity testing algorithm, if the Walkdown fails to embed a back edge to a descendant of c of the biconnected component B with root v^c, then either a $K_{3,3}$ homeomorph can be found based on B, or the K_5 homeomorph based on B can be planarized by removing the internal connections between v and w and replacing B with the cycle (v^c, x, w, y, v^c).*

Proof. Non-planarity minor E has a number of additional tests that can result in the selection of non-planarity minors E_1 to E_4, each of which isolates a $K_{3,3}$ homeomorph. Therefore, we focus on the conditions under which minor E_5 is selected. In minor E_5, the w candidate is externally active, the highest x-y path has points of attachment equal to x and y, and the external connections from x, y and w to ancestors of v have

been selected and at least two are equal and not ancestors of the third.

Case 1: If there is any pertinent or externally active vertex other than w, x and y along the external face path $(x, \ldots, w, \ldots, y)$ of B, then a $K_{3,3}$ homeomorph can be isolated by non-planarity Minor E_1.

Case 2: If there exists any (low) x-y path in B with a point of attachment $p_x \neq x$ or $p_y \neq y$, then a $K_{3,3}$ homeomorph can be isolated by non-planarity minor E_4.

Case 3: If the x-y path in B contains a single endpoint z of a second path p of the form $(w, \ldots, z)$, where all vertices in the path (except w) are embedded inside B, then a $K_{3,3}$ homeomorph can be isolated by the non-planarity minor E_6 depicted in Figure 11.2. This minor is symmetric to non-planarity minor D. The paths represented by edges (u, v), (x, w) and (w, y) are not needed in the $K_{3,3}$ homeomorph.

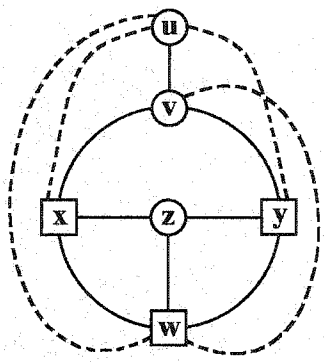


Figure 11.2: Minor E_6

Case 4: Let z be a descendant of u_{max} and an ancestor of v (i.e. comparing DFI, $u_{max} < z < v$). If there exists such a z, and there is an external connection from w to z, then a $K_{3,3}$ homeomorph can be isolated by non-planarity minor E_2. If any such z exists and there is an external connection from x or y to z, then a $K_{3,3}$ homeomorph can be isolated by non-planarity minor E_3.

Case 5: If $u_w < u_{max}$ and there exists a bridge in β_P that straddles u_{max}, then a $K_{3,3}$ homeomorph can be isolated by non-planarity minor E_7. This minor is depicted in Figure 11.3, and it can be isolated by reduction to non-planarity minor C.

Case 6: If $u_y < u_{max}$ and there exists a bridge in β_P that straddles u_{max}, then a

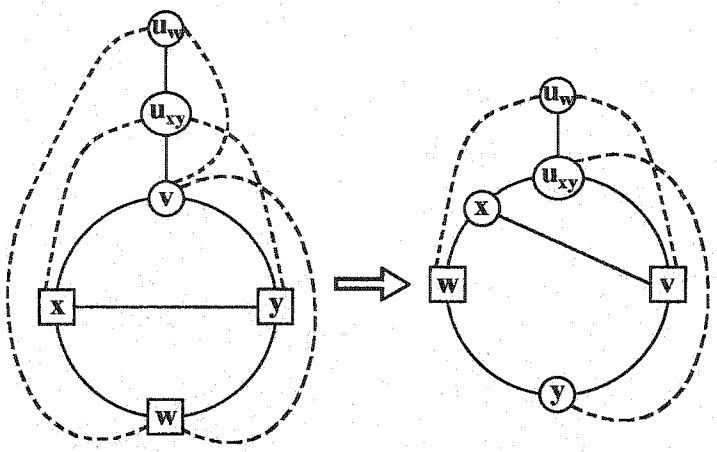


Figure 11.3: Minor E_7 Reduces to Minor C

$K_{3,3}$ homeomorph can be isolated by non-planarity minor E_8, which is shown in Figure 11.4. Note the symmetric case for $u_x < u_{max}$. The paths in the graph corresponding to the edges (v, y), (x, w) and (u_{wx}, v), excluding the endpoints, are not needed to form the $K_{3,3}$ homeomorph. In the symmetric case (in which $u_x < u_{max}$), the edges to omit are (v, x), (y, w) and (u_{wy}, v)).

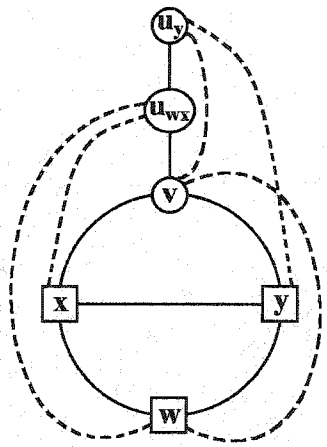


Figure 11.4: Minor E_8

This concludes the cases for which a $K_{3,3}$ homeomorph can be identified based on B, and we now turn to proving that sufficient conditions have been established to warrant planarizing the K_5 homeomorph that has been encountered by removing the internal connections between v and w and replacing B with a cycle (v^c, x, w, y, v^c).

The f_w-bridge of β_{P+B} is planar and connects only to v and w. Thus, since

every path exiting the f_w-bridge must contain v or w, the f_w-bridge can be reduced to a single edge (v, w) due to Theorem 11.4. In fact, in a triconnected component, the internal connections between v and w would already be represented by a single edge. By the same arguments, the x-y path in B can be reduced to a single edge (x, y) due to the failure of the conditions in Cases 2 and 3 and the conditions that resulted in choosing non-planarity minor E_5. Likewise, B can be reduced to a K_4 consisting of the cycle (v^c, x, w, y, v^c) plus the edges (v, w) and (x, y), based solely on the failure of the conditions in Cases 1 to 3, along with the conditions which resulted in choosing minor E_5. In a triconnected component, the failure of the conditions in the six cases above guarantees that B is isomorphic to K_4.

To complete the discussion of the bridges in β_{P+B} but not in β_P, we must consider three possible trivial bridges, direct back edges from x, y and w to ancestors of v, and three possible non-trivial bridges, the F_x-bridge, the F_y-bridge, and the F_w-bridge. A *high bridge* is a bridge of β_{P+B} that attaches to one of x, y or w and at least one ancestor of u_{max}. A *low bridge* is a bridge of β_{P+B} that attaches to one of x, y or w and only u_{max}. Due to the failure of the condition in Case 4, none of the six possible bridges listed above attaches to an ancestor of v that is descendant to u_{max}. Moreover, by virtue of having selected minor E_5, only one of x, y or w can have a trivial or non-trivial high bridge, and the remaining bridges attached to the other two descendants of v in B must attach only to u_{max}.

For each non-trivial low bridge, every path leading into the bridge from a vertex in B must contain u_{max} or the single point of attachment of the bridge to B (namely, x, y or w). Thus, the image vertices of any $K_{3,3}$ homeomorph involving the bridge must be entirely within the bridge, and the $K_{3,3}$ homeomorph is maintained under reductions of B that maintain a path from u_{max} through v to the point of attachment of the bridge to B. The trivial low bridges are of little interest except that, if they exist, then they directly make the desired path connection for the corresponding non-trivial low bridge. In a triconnected component, the low bridges are trivial (represented by a single edge).

If there is a non-trivial high bridge, then due to the failure of the conditions in

Cases 5 and 6, there are no paths from vertices in the bridge to vertices in B that do not include either u_{max} or the single point of attachment of the bridge to B. The corresponding trivial bridge, if it exists, does not affect this argument. Thus, any $K_{3,3}$ homeomorph of the input graph that includes image vertices in the high bridge is cut off at u_{max} and the single point of attachment of the bridge to B such that the $K_{3,3}$ homeomorph is preserved under reductions of B that maintain a path from u_{max} to the point of attachment of the high bridge to B. In a triconnected component, the high bridges are represented by a single edge to u_{max}.

Hence, we are left to consider the non-straddling bridges of β_P. A bridge of β_P attached to the path $(v, \ldots, u_{max})$ can only connect to vertices of B by paths containing v or u_{max}, so any $K_{3,3}$ homeomorph of the input graph that includes image vertices from such a bridge is cut off at u_{max} and v from including more than a path through B leading from v through any descendant of v and back up to u_{max}. In a triconnected component, the path P is represented by a single edge from v to u_{max} such that the entire input graph is isomorphic to K_5 and thus contains no $K_{3,3}$ homeomorph.

Finally, any bridge of β_P attached to the path $(u_{max}, \ldots, u_{min})$ can only connect to B by paths passing through u_{max} and the vertex of B (x, y or w) with an external connection to an ancestor of u_{max}. Once again, any $K_{3,3}$ homeomorph with image vertices in such a bridge is maintained under a reduction of B provided that a path from u_{max} through v to the point of attachment of the high bridge to B. For triconnected components, these bridges are represented by the same edge that represents the high bridges.

As a summative result, any $K_{3,3}$ homeomorph of the input graph has at most one image vertex in B, which is shared by the bridge(s) representing the $K_{3,3}$ homeomorph, such that B provides at most one path to the $K_{3,3}$ homeomorph. Since the cycle (v^c, x, w, y, v^c) maintains a path in B from v^c to each of x, y and w, the aforementioned edges (v, w) and (x, y) can be deleted, restoring the planarity of the portion of the input graph represented by the embedding structure such that it can continue the search for $K_{3,3}$ homeomorphs elsewhere in the graph. □

Corollary 11.6 *A non-planar triconnected component of a graph is isomorphic to K_5 or contains a $K_{3,3}$ homeomorph.*

Exploiting Theorem 11.5 to create a linear time $K_{3,3}$ search algorithm is remarkably easy because only a few additional tests are needed beyond those that initially select non-planarity minor E_5. If any test succeeds, then a $K_{3,3}$ homeomorph can be isolated. If all of the tests fail on a triconnected component, then the triconnected component is isomorphic to K_5 and therefore contains no $K_{3,3}$ homeomorph. If the input graph is not triconnected, then linear time performance can be maintained in most cases, as described below.

Even with initialization restricted to the biconnected component B with root v^c, the normal tests in the Kuratowski subgraph isolator for minors A to D (in Lines 3 to 17 of procedure ChooseTypeOfNonplanarityMinor, Figure 9.3) and E_1 to E_5 (in Lines 1 to 5 of procedure IsolateMinorE, Figure 9.17) can be performed without difficulty because they are only based on information stored in the vertex structures and edge records that represent B. If the result of the tests is minor E_5, then the case logic of Theorem 11.5 must be applied.

The test for Case 1 can be implemented using a slightly modified form of procedure FindExtActivityBelowXYPath from Section 9.2, which operates identically except for skipping w.

The test for Case 2 of Theorem 11.5 can be implemented with the help of a slightly modified form of procedure MarkHighestXYPath from Section 9.3, which operates identically except for returning with $p_x = p_y = nil$ if w is encountered while searching for an x-y path. To test whether there exists an x-y path with points of attachment below x or y, we invoke the modified procedure twice. First, we hide the internal edges of x, then invoke the procedure. If it succeeds, then we have an x-y path with a point of attachment p_x below x. If it fails, then we restore the edges of x, then hide the internal edges of y and invoke the procedure again. If it succeeds, then we have an x-y path with a point of attachment p_y below y. Otherwise, we restore the edges of y and try the next case.

The test for Case 3 of Theorem 11.5 can be implemented as follows. First, hide

all internal edges of vertices along the external face path $(x, \ldots, w, \ldots, y)$ except those of w and the two edges that attach x and y to the x-y path. In this way, a walk of the proper face that starts with the path $(w, \ldots, x)$ returns to w before reaching y if and only if the internal w-z path required in minor E_6 exists. The procedure for this is similar to procedure MarkVtoZPath from Section 9.3, with the principal exception that it must ignore paths whose points of attachment are both on the x-y path (in a fashion similar to procedure MarkHighestXYPath of Section 9.3).

The test for Case 4 of Theorem 11.5 can be performed by examining the vertices along the path from v to u_{max}, excluding the endpoints. It is helpful to associate with each vertex the maximum numbered DFS descendant, which can easily be computed during preprocessing. For each vertex z on the path $(v, \ldots, u_{max})$ and distinct from the endpoints, examine the back edges of z to see whether any lead to the subtrees rooted by x, y or w. For a back edge from z to a descendant d_z, the subtree rooted by a vertex x contains d_z if d_z is in the range DFI(x) to MaxDescendant(x). Note that two of x, y and w descend from the third, but once we know that a $K_{3,3}$ homeomorph exists, it takes little extra effort to resolve whether d_z is x, y, w or which of those has the separated child biconnected component leading to d_z.

The tests for Cases 5 and 6 of Theorem 11.5 are implemented by traversing the path $(v, \ldots, u_{max})$. At each vertex z, we test for a connection to an ancestor of u_{max} by taking the minimum of the leastAncestor of z and the least lowpoint of the DFS children of z, excluding the DFS child of z that is equal to or an ancestor of the child c in the root edge with v^c (i.e. use the second element of the separatedDFSChildList of z if it exists and if the first element is the child to be excluded). If the computed minimum is less than the DFI of u_{max}, then a straddling bridge has been found.

Theorem 11.7 *Given a simple undirected graph G with n vertices, a subgraph homeomorphic to $K_{3,3}$ can be found or shown not to exist in G in $O(n)$ time.*

Proof. The fact that a $K_{3,3}$-less graph has $3n - 5$ or fewer edges is a result of Asano [4]. Thus, our data structure limit to kn edges with $k \geq 3$ is applicable. The triconnected components of a graph can be obtained in $O(n)$ time [28, 56]. The cost

of identifying root copies of v on which the Walkdown terminated with Condition 1 or 2 is commensurate with the cost of the Walkup operations performed in step v. For a given root v^c of a biconnected component B, the determination of minor A, B, C, D, E_1, E_2, E_3, E_4, or E_5, including the identification of the highest x-y path when required by the selected minor, have costs linear in the number of vertices in B. Since B either becomes part of a $K_{3,3}$ homeomorph or is reduced to a constant size, the $O(n)$ total run-time is maintained.

The tests for Cases 1 to 3 of Theorem 11.5 also have costs linear in the size of B. The test for Case 4 is commensurate with the sum of the degrees of vertices on the DFS tree path strictly between v and u_{max}. If the test succeeds at finding the desired connection, then a $K_{3,3}$ homeomorph can be isolated so $O(n)$ run-time is maintained. The test for Cases 5 and 6 is linear in the number of vertices on the DFS tree path $(v, \ldots, u_{max})$. If the tests for Cases 4 to 6 fail, then v and u_{max} are a separation pair for the DFS tree path $(v, \ldots, u_{max})$, and would therefore be represented by a single edge (v, u_{max}) in a triconnected component, such that the tests for Case 4 to 6 take only constant time. □

Remark: If the input graph is not triconnected, but only a predetermined constant number of K_5 homeomorphs use vertices along the path $(v, \ldots, u_{max})$, then $O(n)$ performance can be maintained by reducing B to a constant size if a $K_{3,3}$ homeomorph is not found when a Condition 2 Walkdown failure occurs on B. Since it is easy to detect whether edges or vertices are visited more than a predetermined constant number of times when processing an input graph, the need to isolate triconnected components can be obviated when applying the $K_{3,3}$ isolator in contexts where a graph cannot contain too many obstructions to planarity.

12. Future Work

This dissertation provides new linear time algorithms for planarity testing, planar embedding and Kuratowski subgraph isolation. These are the three fundamental combinatorial algorithms relating to graph planarity, for with them one can both determine whether a graph is planar and obtain a simple 'certificate' showing why the graph was deemed planar or non-planar. This dissertation also contributes new linear time algorithms to solve the corresponding problems for outerplanarity, as well as new linear time algorithms that search for $K_{2,3}$ and $K_{3,3}$ homeomorphs. Still, there exists a plethora of future research opportunities, due in equal parts to the maturity of this field of study and to the relative simplicity of the new algorithms and data structures in this dissertation. This chapter discusses a number of these possibilities.

12.1 Drawing Planar Graphs as HorVert Diagrams

A combinatorial planar embedding of a graph G provides enough information to allow a subsequent face-walking algorithm to list the bounding walk of each face in any planar representation of G. However, it does not provide vertex positions and edge shapes for a specific planar representation of G. A number of algorithms listed in Chapter 1 do not require the conversion of combinatorial planar embeddings to specific planar representations. Moreover, the desired planar representation for a given combinatorial planar embedding can be application-specific. For these reasons, the generation of a specific planar representation is typically considered to be a separate problem.

Nevertheless, the combinatorial planar embedding as answer to the question 'Is G planar?' can seem obscure, and as such there is some merit in producing the vertex positions and edge shapes of a specific planar representation of G if it is planar. A *HorVert diagram* 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. By expanding the geometric representation of a vertex horizontally, the difficulty of

selecting an edge shape is reduced to two smaller problems: choosing a horizontal vertex length and choosing a horizontal edge position. Another advantage of HorVert diagrams is that they can be drawn with the simplest possible rendering capabilities because it is composed solely of horizontal and vertical lines, plus any symbols needed to label the vertices. The prospects for augmenting our new algorithms with operations that result in vertex positions and lengths and edge positions seem high due to the prior work of Jayakumar, Thulasiraman and Swamy [32], which presents augmentations for the PQ-tree operations of Booth and Lueker [7] that help to generate a HorVert diagram for the given planar graph.

However, the algorithm of Jayakumar, Thulasiraman and Swamy [32] is critically dependent on the property of an st-numbering that it has a single source (namely, s). Their method creates a τ order for each vertex i, which is the edge order of previously embedded vertices numbered less than i. Moreover, it separates the τ order into three partitions, denoted τ_L, τ_C and τ_R, based on what was happening in the PQ-tree at the time vertex i was embedded. When the PQ-tree T_{i-1} is converted to T^*_{i-1} by performing permutations and reflections such that all leaves containing i are consecutive, we consider the children of the subtree root that contains all leaves labeled i. There is at most one partial node on the left, zero or more full nodes, then one partial node on the right. The leaves labeled i are partitioned into τ'_L, τ'_C and τ'_R based on these nodes. When the embedding is finished, the three partitions τ_L, τ_C and τ_R for a vertex can be recovered easily from τ'_L, τ'_C and τ'_R. The reason for this is that, when vertex i is added, all of its incident edges are bound into a single biconnected component in the partial embedding. Otherwise, we contradict the assertion that the graph has a single source. Because all edges in τ'_L, τ'_C and τ'_R are all in a single biconnected component, the worst that can happen during the remainder of the embedding is that they get reversed, which can be accommodated by assigning the reverse of τ'_L to τ_R, the reverse of τ'_C to τ_C and the reverse of τ'_R to τ_L.

The dependence on a single source is problematic with respect to our new algorithms since every leaf of the DFS tree is a source. The solution to this problem may rest in finding an alternative formulation of their method to our data structures that

eliminates this dependence.

12.2 Testing for the Consecutive Ones Property

A (0, 1)-matrix M 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. The PQ-tree data structure was created by Booth and Lueker as the primary component in an efficient test for the consecutive ones property. According to Booth and Lueker [7, p. 372-3], it was Tarjan who first suggested the possible application of the PQ-tree to planarity based on similarities between PQ-tree operations and the formula manipulation operations of the Lempel, Even and Cederbaum planarity testing algorithm [42]. In turn, our new data structures and our methods for updating them have numerous similarities with PQ-trees and their operations in the context of solving a graph planarity problem. These similarities suggest the possibility that our data structures and algorithms may be a suitable alternative to complex PQ-tree template application, pruning and reduction for testing the consecutive ones property. The solution to this problem may rest in having the ability to create a graph that corresponds to a given (0, 1)-matrix such that testing the planarity of the graph by our new method is equivalent to testing the consecutive ones property of the corresponding matrix. On the other hand, a proof that no such correspondence exists would be equally valuable.

12.3 Triconnectivity in Planar Graphs

Linear time algorithms for determining triconnectivity in planar graphs are due to Hopcroft and Tarjan [28] and Vo [56]. Separation of a graph into triconnected components helps achieve linear time for $K_{3,3}$ search algorithms such as the one in this dissertation and the one due to Fellows and Kaschube [20]. As well, the identification of triconnected components and their planar embeddings is a useful first step in well-known planar graph isomorphism testing algorithms due to Hopcroft and Tarjan [29] and Hopcroft and Wong [31].

Our planar embedding algorithm and our data structures seem well-suited to

simple augmentations that would result in an alternative, possibly linear time, approach to triconnectivity. Our data structure maintains the biconnected components that develop as edges are added from the input graph to the embedding. Our algorithm embeds each back edge e individually, and when the edge joins vertices in separate biconnected components, two or more biconnected components may be merged such that, together with e, a single biconnected component results. Prior to the merge, a biconnected component is separated from its parent by a cut vertex r. When a biconnected component B is merged to form a larger biconnected component, B is attached by a separation pair consisting of the vertex r and some other vertex w, which either was a cut vertex to a child biconnected component or is the descendant endpoint of the back edge e. An augmentation to our data structures would typically not need to maintain information about B if no other back edge results in the attachment of an edge or child biconnected component along the external face path between w and r.

However, there are some additional cases to consider. For instance, a biconnected component B rooted by vertex r and containing a vertex w is merged into the biconnected component containing the current vertex v when the algorithm embeds the back edge (v, w), but r and w are not the only resulting separation pair. The pair v and r are also a separation pair, and this does not change if the embedding of more back edges from v results in attachment of edges or child biconnected components along the external face path between w and r.

12.4 Enumerating, Ranking and Unranking Planar Embeddings

A triconnected planar graph G has only two planar embeddings, $\tilde{G}_1$ and $\tilde{G}_2$, which are equivalent except for inverted edge orientation of each vertex. Thus, flipping $\tilde{G}_1$ on the plane yields $\tilde{G}_2$, and vice versa. Matters become more complicated for graphs that are not triconnected. The biconnected components attached to a cut vertex and the triconnected components attached to each separation pair can be

arbitrarily permuted and flipped.

It should be possible to define a total ordering over all planar embeddings given one planar embedding of a graph G produced by our algorithms, possibly augmented by the work in Section 12.3 to identify separation pairs. A permutation generator could be used to iterate through the available configurations at each cut vertex and the lower numbered vertex in each separation pair. Also, a bit string could be used to track the flipping of biconnected and triconnected components. For example, the bit for a biconnected component could be 0 if the link[0] member of the biconnected component root indicated an edge to a lower numbered neighbor than the link[1] member, and 1 otherwise.

Thus, given a planar embedding $\tilde{G}$ of a graph G, determining the rank of $\tilde{G}$ would be a matter of assessing the permutations in evidence at each cut vertex and at the lesser vertex of each separation pair, as well as assessing the bit string corresponding to the flip state of each biconnected and triconnected component. Finding the planar embedding of least rank relative to $\tilde{G}$ is a matter of permuting and flipping biconnected and triconnected components such that the least bit string and permutation is in evidence at each cut vertex and separation pair. Enumeration of planar embeddings can proceed from the least ranked planar embedding relative to the planar embedding $\tilde{G}$ produced by our algorithms by systematically enumerating through the aforementioned bit strings and permutations. The work of flipping components can be kept to a minimum using a binary reflected Gray code sequence and can likely be kept to constant time using a technique similar to but more sophisticated than the edge signing used in our planarity algorithm. The work of permuting components can be minimized using a single transposition (Gray code) permutation generator. The triconnected components mapped to the transposed permutation elements can be moved in constant time if the adjacency lists are doubly linked and if sufficient information exists in the data structure to easily find the sequence of edges that attach each component to its cut vertex or separation pair.

Unranking a number n to produce the n^{th} planar embedding relative to the initial planar embedding $\tilde{G}$ produced by our algorithms is also possible. By using

the well-known Trotter-Johnson algorithm as the permutation generator, a straightforward permutation unranking algorithm can be used to match the Trotter-Johnson algorithm's generation sequence [40].

One limitation of the approach described in this section is that it is dependent on the actions of the initial depth first search of a graph G, which is performed in the initial steps of generating the planar embedding $\tilde{G}$. Two isomorphic graphs with non-equivalent vertex orders or adjacency list edge orders are processed differently by depth first search. This in turn affects the planar embedding produced by our algorithms and ultimately the defined total ordering upon which the above enumeration, ranking and unranking ideas are based.

12.5 Searching for K_5 Minors and Homeomorphs in Linear Time

Searching for K_5 minors and homeomorphs in linear time is a long-standing open problem. In fact, the best related work currently known is the $O(n^2)$ algorithm of Kézdy and McGuinness [37] for isolating a K_5 minor. The difficulty of the problem is best stated by Fellows and Kaschube: "We have attempted to modify the approach behind the first algorithm [for $K_{3,3}$ homeomorph isolation] to provide a practical algorithm for the K_5 subgraph homeomorphism problem, but there seem to be no simple analogs to Theorems 1 and 2" [20, p. 289]. The new $K_{3,3}$ homeomorph isolator in this dissertation is much more closely aligned to finding $K_{3,3}$ homeomorphs in the context of the properties of the depth first search tree at the time non-planarity is discovered. Hence, the new approach may yield significant insights into solving the K_5 subgraph homeomorphism problem, particularly if the 4-connected components of a graph are considered separately.

12.6 Embedding in the Projective Plane

The *projective plane* is equivalent to a disk with antipodal points identified. Although Mohar [46] created a linear time algorithm for embedding graphs in the

projective plane, it is exceedingly complex [64]. Myrvold and Roth [48] created a simpler algorithm exploiting some of the same properties, though it does not achieve linear time. Thus, a projective planar embedding algorithm that is both relatively simple and linear time is wanting.

A projective planar embedding less the edges that pass through antipodal point identifications is a planar embedding in which all endpoints of the edges so removed are on the external face. Our new algorithms and data structures are specially suited to maintaining the external face and embedding edges one at a time. Projective planarity therefore appears to be a promising area of future research involving the new algorithms in this dissertation. Given that there are a limited number of embeddings of a Kuratowski subgraph [46] and that the adjacency lists can be reordered such that edges which must pass through antipodal point identifications are back edges, the solution to this problem may rest in the ability to find all of the edges to remove and maintaining their endpoints on the external face.

Bibliography

[1] K. Appel and W. Haken. Every planar map is four colorable. Part I. Discharging. *Illinois Journal of Mathematics*, 21:429–490, 1977.

[2] K. Appel and W. Haken. *Every planar map is four colorable.* Contemporary Mathematics, Volume 98. American Mathematical Society, Providence, RI, 1989.

[3] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. Part II. Reducibility. *Illinois Journal of Mathematics*, 21:491–567, 1977.

[4] T. Asano. An approach to the subgraph homeomorphism problem. *Theoretical Computer Science*, 38:249–267, 1985.

[5] L. Auslander and S. V. Parter. On imbedding graphs in the plane. *Journal of Mathematics and Mechanics*, 10:517–523, 1961.

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

[7] 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.

[8] 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.

[9] W. M. Brehaut. An efficient outerplanarity algorithm. *Congressus Numerantium*, 26:99–113, 1977.

[10] J. Cai. Counting embeddings of planar graphs using DFS trees. *SIAM Journal of Discrete Mathematics*, 6(3):335–352, 1993.

[11] 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.

[12] N. Chiba, T. Nishizeki, and N. Saito. A linear 5-coloring algorithm of planar graphs. *Journal of Algorithms*, 2(4):317–327, 1981.

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

[14] G. Demoucron, Y. Malgrange, and R. Pertuiset. Graphes planaires. *Rev. Française Recherche Opérationnelle*, 8:33–47, 1964.

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

[16] J. A. Ellis, M. Mata, and G. MacGillivray. A linear time algorithm for longest (s, t)-paths in weighted outerplanar graphs. *Information Processing Letters*, 32:199–204, 1989.

[17] L. Euler. Demonstratio nonnullarum insignium proprietatum quibus solida hedris planis inclusa sunt praedita. *Novi Comm. Acad. Sci. Imp. Petropol*, 4:140–160, 1758.

[18] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

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

[20] M. R. Fellows and P. A. Kaschube. Searching for $K_{3,3}$ in linear time. *Linear and Multilinear Algebra*, 29:279–290, 1991.

[21] T. Feo and J. Provan. Delta-wye transformations and the efficient reduction of two-terminal planar graphs. *Operations Research*, 41(3):572–582, 1993.

[22] G. Frederickson. On linear-time algorithms for five-coloring planar graphs. *Information Processing Letters*, 19:219–224, 1984.

[23] M. Garey and D. Johnson. *Computers and Intractability: A Guild to the Theory of NP-completeness.* Freeman, 1979.

[24] Alan Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[25] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Graph and Combinatorics Conf.* Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dep.of Math., Princeton U., 2 pp., 1963.

[26] F. Hadlock. Finding a maximum cut in a planar graph in polynomial time. *SIAM Journal of Computing*, 4(3):221–225, 1975.

[27] J. Hopcroft and R. Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Information Processing Letters*, 1:32–34, 1971.

[28] J. Hopcroft and R. Tarjan. Dividing a graph into triconnected components. *SIAM Journal of Computing*, 2:135–158, 1973.

[29] J. Hopcroft and R. Tarjan. A $V\log V$ algorithm for isomorphism of triconnected planar graphs. *Journal of Computer and Systems Sciences*, 7:323–331, 1973.

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

[31] J. Hopcroft and J. Wong. Linear time algorithm for isomorphism on planar graphs. *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[32] 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.

[33] 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.

[34] M. Juvan, J. Marincek, and B. Mohar. Embedding a graph in the torus in linear time. In *Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science, Vol. 920*, pages 360–363. Springer, 1995.

[35] A. Karabeg. Classification and detection of obstructions to planarity. *Linear and Multilinear Algebra*, 26:15–38, 1990.

[36] A. Karabeg. Ranking planar embeddings using *PQ*-trees. *Annals of Discrete Mathematics*, 55:249–260, 1993.

[37] A. Kézdy and S. McGuinness. Sequential and parallel algorithms to find a K_5 minor. *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 345–356, 1992.

[38] P. Klein and J. Reif. An efficient parallel algorithm for planarity. 27^{th} *Annual IEEE Symposium on Foundations of Computer Science*, pages 465–477, 1986.

[39] W. Klotz. A constructive proof of Kuratowski's theorem. *Ars Combinatoria*, 28:51–54, 1989.

[40] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search.* CRC Press, 1999.

[41] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[42] 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.

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

[44] D. Matula, Y. Shiloach, and R. Tarjan. Two linear-time algorithms for five-coloring a planar graph. Technical Report CS-80-830, Department of Computer Science, Stanford University, Nov. 1980.

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

[46] B. Mohar. Projective planarity in linear time. *Journal of Algorithms*, 15:482–502, 1993.

[47] B. Mohar. Embedding graphs in an arbitrary surface in linear time. *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 392–397, 1996.

[48] W. Myrvold and J. Roth. Simpler projective plane embedding. *Sixth International Conference on Graph Theory, Marseille France*, pages 1–4, 2000.

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

[50] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. A new proof of the four-colour theorem. *Electronic Research Announcements of the American Mathematical Society*, 2(1):1–9, 1996.

[51] F. Rubin. An improved algorithm for testing the planarity of a graph. *IEEE Transactions on Computers*, c-24(2):113–121, 1975.

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

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

[54] R. Thomas. An update on the four-color theorem. *Notices of the American Mathematical Society*, 45(7):848–859, 1998.

[55] W. T. Tutte. Matroids and graphs. *Trans. Amer. Math. Soc.*, 88:144–174, 1958.

[56] K. P. Vo. Finding triconnected components of graphs. *Linear and Multilinear Algebra*, 13:143–165, 1983.

[57] K.P. Vo, W. Dick, and S. G. Williamson. Ranking and unranking planar embeddings. *Linear and Multilinear Algebra*, 18:35–65, 1985.

[58] K. Wagner. Über einer eigenschaft der ebener complexe. *Math. Ann.*, 14:570–590, 1937.

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

[60] R. Wiese. *Personal Communication.* June 6, 2001.

[61] S. G. Williamson. Embedding graphs in the plane- algorithmic aspects. *Ann. Disc. Math.*, 6:349–384, 1980.

[62] S. G. Williamson. Depth-first search and Kuratowski subgraphs. *Journal of the Association for Computing Machinery*, 31(4):681–693, 1984.

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

[64] S. G. Williamson. *Math Reviews*, 94f:05141, 1994.