Trinity University
Digital Commons @ Trinity

Computer Science Honors Theses   Computer Science Department

4-21-2009

A New Parallel Algorithm for Planarity Testing

Rebecca Ingram
Trinity University

Follow this and additional works at: http://digitalcommons.trinity.edu/compsci_honors

🅖 Part of the Computer Sciences Commons

Recommended Citation

A New Parallel Algorithm for Planarity Testing

Rebecca Ingram

Abstract

Determining whether a graph is planar is both theoretically and practically interesting. Although several sequential algorithms have been introduced which accomplish planarity testing in $O(V)$ time for graphs with V vertices, very few of these have been parallelized. In a recent comparison of sequential planarity testing algorithms, the newest algorithms were found to be fastest; however, these are the ones which have not been parallelized. The goal of this thesis is to introduce a method for parallelizing one of the newest planarity testing algorithms.

Acknowledgments

The author would like to thank the following people for their support during the writing of this thesis:

Berna Massingill my thesis advisor, for considerable help with the revision and clarification of these ideas

Maurice Eggen and Paul Myers my committee, for providing helpful feedback

Cameron Swords for his continual encouragement and willingness to discuss these ideas

A New Parallel Algorithm for Planarity Testing

Rebecca Ingram

A departmental thesis submitted to the

Department of Computer Science at Trinity University

in partial fulfillment of the requirements for Graduation

with departmental honors.

April 21, 2009

_____   _____

Thesis Advisor   Department Chair

_____

Associate Vice President

for

Academic Affairs

A New Parallel Algorithm for Planarity Testing

Rebecca Ingram

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the development of multi-core and multi-processor systems that are capable of completing more and more tasks in parallel, it is increasingly important to develop algorithms which can take full advantage of these capabilities. Some tasks, however, have been found to be more easily parallelizable than others. For example, if one wanted to add a list of numbers, one could simply assign equal portions of the list to each process or thread, compute the sums of these smaller lists, and then add all of the results together.

In contrast, the parallelization of graph algorithms is much less straightforward. Perhaps a large part of the difficulty lies in the fact that graphs have a kind of inherent irregularity in that each vertex may be adjacent to any number of other vertices in the graph. Furthermore, there is not always a pattern in these connections between vertices. These "irregularities" contribute to difficulties in dividing the work into subtasks so that the workloads are well-balanced.

One such problem is planarity testing. A graph is planar if it may be drawn in such a

way that no pair of edges intersects. In addition to being of theoretical interest, determining whether a given graph is planar also has practical applications. For example, in VLSI it is important to determine whether a circuit diagram is planar because an edge crossing will cause a short circuit. Furthermore, according to Garey and Johnson [6], many instances of NP-complete problems, such as finding cliques, bipartite subgraphs, max cut, feedback arc sets, and graph isomorphisms may be solved in polynomial time if the input graph is planar.

Several sequential algorithms have been proposed to solve the planarity testing problem. The algorithm introduced by Hopcroft and Tarjan [8] was the first to accomplish planarity testing in $O(V)$ time, where V is the number of vertices in the graph. Later, Booth and Lueker described the PQ-tree data structure, which can be used as part of a planarity testing algorithm. More recently, Shih and Hsu [15, 9] introduced the PC-tree data structure, which is simpler and faster than PQ-trees. Boyer and Myrvold [5] described a different algorithm based on construction of biconnected components, which is considered by some to be identical to the algorithms used with PC-trees [7].

Although several sequential algorithms exist which can test for planarity in $O(V)$ time, only a few authors seem to have addressed the problem of parallelizing planarity testing algorithms. Klein and Reif [11] describe parallel algorithms for the PQ-tree data structure (which will be further discussed later), and Bader and Sreshta [1] identify errors in [11] and other algorithms based on it and provide corrections for them. Ramachandran and Reif [14] describe a parallel planarity testing algorithm based on open ear decomposition.

The following sections provide the terminology needed to understand how planarity testing is done, introduce a few sequential planarity testing algorithms to familiarize the reader with some commonly used techniques, describe Boyer and Myrvold's edge addition method, and finally lead up to the proposal of a parallelization of Boyer and Myrvold's

algorithm.

1.2 Terminology

In order to prevent confusion, the definitions of the graph-related terms that will be used throughout are provided. Readers already familiar with these concepts are invited to skip ahead to Chapter 2.

Readers are assumed to be familiar with the standard definition of a graph (a set of vertices and edges), as well as the meanings of connected components, cycles, and subgraphs. Throughout, assume that V represents the number of vertices in a graph, and that E is the number of edges.

A graph is planar if it may be drawn in a plane such that no pair of edges intersects; this planar representation is called an embedding. One point that should be discussed here is how embeddings can be represented and differentiated from each other. Although a graph can be completely described as a set of vertices and edges, this is insufficient for embeddings. To describe an embedding, the order in which edges are arranged around a vertex is used. This ordering is called the orientation of a vertex. Edges may be listed in either clockwise or counterclockwise order, as long as the use is consistent throughout. Notice that reversing the orientation of all vertices (that is, switching from a clockwise order to a counterclockwise order or vice versa) produces a graph which is a mirror image of the original. Reversing the orientation of the vertices does not affect whether or not an embedding is planar, and PQ-trees, PC-trees, and the Boyer-Myrvold edge addition algorithm all take advantage of this fact.

Another important concept is homeomorphism. McConnell and Hsu [10] describe this idea in terms of subdivision. The subdivision of an edge with endpoints a and b is created

by adding a new vertex, say f, deleting the edge (a, b), and inserting the two new edges (a, f) and (f, b). A more concise definition is provided in [5], which states that a graph is homeomorphic to another graph if it is identical except that paths may replace some of the edges. Refer to Figure 1.1 for an example.
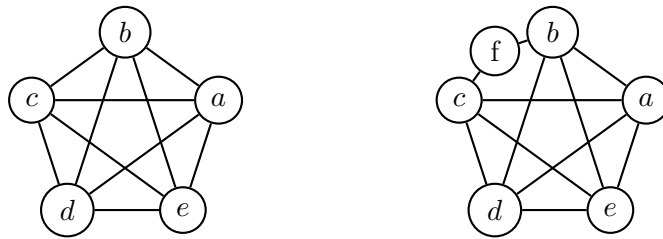

Figure 1.1: A pair of homeomorphic graphs

Another important concept is that of biconnected components. Hopcroft and Tarjan provide an excellent explanation of what they are. Given any pair of vertices, x and z, if there is a third vertex y such that every path from x to z includes y, then y is a cut vertex (also known as an articulation vertex), and the graph is not biconnected. Although they do not discuss methods for doing so, [8] mention that it is possible to divide any graph that is not biconnected into biconnected components in linear time. For this reason, some planarity testing algorithms begin with the assumption that the input graph is biconnected. Figure 1.2 demonstrates examples of biconnected and non-biconnected graphs.

Theorem 1.2.1. *A graph is planar if and only if its biconnected components are planar.*

One of the foundational ideas in studies of planar graphs is Euler's formula, which defines the relationship between the number of vertices, edges, faces, and connected components of a graph. The faces of a graph are the regions of the plane which are separated by cycles of the graph. The external face is the region typically considered to be "outside" the graph. In Figure 1.2(b), the external face is defined by the cycle consisting of vertices a, c, d, b,
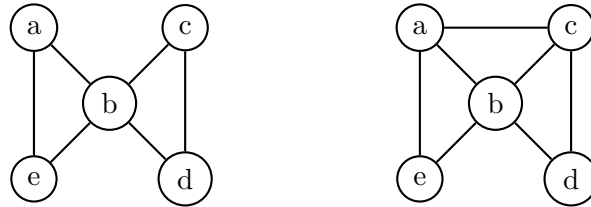
Figure 1.2: A pair of graphs to illustrate biconnected components. The graph on the left is not biconnected. Vertex b is a cut vertex. The graph on the right is biconnected. Every graph may be divided into biconnected components. In the graph of the left, the biconnected components are the subgraphs consisting of b, e, a and b, c, d.

and e. In order for Euler's formula to be correct, the external face must be included in the set of faces. According to [10], Euler's formula states that:

Theorem 1.2.2. *If F is the number of faces and C the number of connected components, then $V + F = E + C + 1$.*

This formula holds for all graphs and is used to derive several important properties of planar graphs. For example, it has been shown that:

Theorem 1.2.3. $E \leq 3V - 6$

The interested reader is referred to [13] for a proof. The fact that the maximum number of edges in a planar graph is proportional to the number of vertices means that the number of edges is $O(V)$. This makes it possible to bound the amount of time required for planarity testing to $O(V)$.

Perhaps the most significant theorem regarding planar graphs is Kuratowski's Theorem, which states that

Theorem 1.2.4. *A graph is planar if and only if it does not contain a subgraph homeomorphic to K_5 or $K_{3,3}$.*

K_5 is the complete graph (every vertex is adjacent to every other vertex) with five vertices, and $K_{3,3}$ is the complete bipartite graph with three vertices. A bipartite graph is one is which the vertices may be divided into two disjoint sets such that the vertices of one set are only adjacent to vertices in the other set. A complete bipartite graph is one in which every vertex of the first set is adjacent to all vertices in the second set and vice versa. Refer to Figure 1.3 for pictorial representations of the two Kuratowski subgraphs.

(a) K_5 (b) $K_{3,3}$
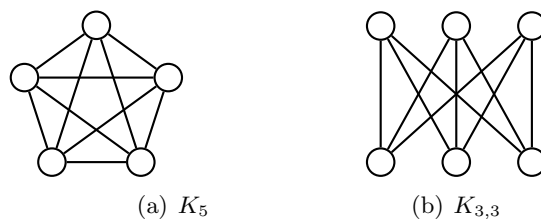
Figure 1.3: Every non-planar graph contains a subgraph homeomorphic to one of the Kuratowski subgraphs. K_5 is the complete graph of five vertices and $K_{3,3}$ is the complete bipartite graph of six vertices.

Although many algorithms for planarity testing include techniques for identifying Kuratowski subgraphs, they usually do not work by searching for these subgraphs, a task which could easily require time proportional to V^6, if not longer [8]. Several common approaches to planarity testing will now be discussed.

Chapter 2

Background and Related Work

2.1 Hopcroft and Tarjan

The first linear time algorithm for planarity testing was developed by Hopcroft and Tarjan [8]. Unlike many later planarity tests, Hopcroft and Tarjan use a path addition approach, which identifies the cycles in a graph and embeds these one at a time. Although their method is very different from subsequent planarity testing algorithms and has also been shown to be the slowest in a comparison of implementations performed by Boyer, et al. [12], three of the ideas discussed in their paper have been used in several other algorithms, and it is for this reason that their techniques will be introduced here.

The first of these three concepts is that of using a depth-first search (DFS) to explore the graph and assign labels to vertices systematically. One advantage of this is that it ensures that all vertices have a sensible label, i.e., vertices are numbered consecutively from 1 to V. More importantly, it guarantees that the labels have certain helpful properties. For example, all vertices in the same connected component have a path to vertex 1 consisting solely of lower-numbered vertices. A recursive definition of DFS is given in Figure 2.1. The

invocation `Depth-First-Search`$(a, 1)$ will assign numbered labels to all vertices beginning with vertex a. To number the vertices in reverse order, simply change the recursive call in line 4 to use $n - 1$ instead of $n + 1$, and use `Depth-First-Search`(a, V) for the initial invocation.

Figure 2.1: Pseudo-code for a depth-first search

Depth-First-Search(v, n)
```
1   for w ∈ neighbors(v)
2         do if w is unlabeled
3              then label(w) ← n
4                   Depth-First-Search(w, n + 1)
```

The second of the ideas used by almost all planarity testing algorithms is the depth-first search tree (DFS tree), which is almost identical to the structure which [8] refers to as a palm tree. The term DFS tree will be used since it is more common in recent literature. The tree is constructed such that the children of a vertex are those vertices for which that vertex calls `Depth-First-Search()` in the algorithm described above. The edges between parents and children in the DFS tree are called tree edges and correspond to arcs in palm trees. The remaining edges are called back edges and correspond to palm tree fronds. Typically, the edges of the tree are considered to be directed: tree edges are directed from the parent to the child, and back edges are directed from the descendant to its ancestor. Figure 2.2 depicts a graph and the corresponding DFS tree.

A third important idea used in [8] is that of lowpoints. The lowpoint of a vertex, v, is the lowest numbered vertex reachable by a single back edge from v or one of its descendants. In Figure 2.2, the lowpoint of vertex 5 is 2; however, if an edge were added between vertices 1 and 6, then the lowpoint of 5 would be 1 because vertex 6 is a descendant of 5 in the DFS
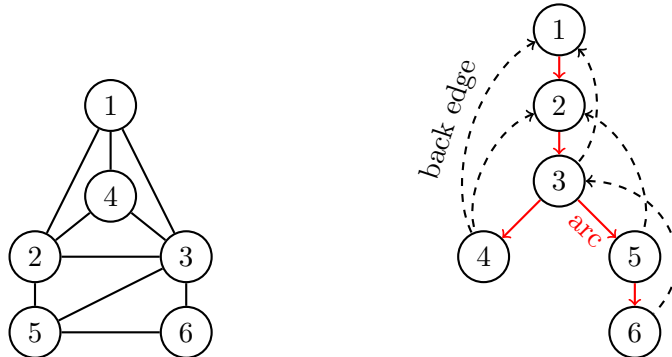
Figure 2.2: A graph (left) with vertices labeled according to a DFS and the corresponding DFS tree (right). In the DFS tree, solid arrows are tree arcs, and dashed arrows are back edges. Arrowheads indicate the direction imposed on the edge.

tree.

To determine whether a graph is planar, Hopcroft and Tarjan first split up the graph into its biconnected components and then use a DFS to explore each of these components and construct a DFS trees. A recursive pathfinding algorithm is used to identify one cycle and several paths. Note that because the edges of the DFS tree are assumed to have a direction, the cycle must consist of two or more tree arcs and exactly one back edge. The cycle is removed from the graph, dividing it into smaller segments. To test for planarity, an attempt is made to construct a planar embedding, beginning with the cycle. Every segment that is added to the embedding must be inserted either inside or outside the cycle. If a conflict occurs, segments may be moved to the other side of the cycle; however, if a segment cannot be inserted, then the graph is known to be non-planar. To ensure that the algorithm can run in $O(V)$ time, two stacks named L and R are maintained to keep track of the side of the cycle on which back edges are embedded. A block is defined as the maximal set of back edges in L and R such that "the placement of any one of the [back edges] determines the placement of all the others" [8]. Using the blocks and stacks, it can be shown that the

algorithm runs in $O(V)$ time.

This method has a several disadvantages. First, the implementation details are somewhat cumbersome: two stacks and several blocks all have to be maintained and updated. Furthermore, no method is given for isolating Kuratowski subgraphs, which means that the algorithm only states whether a graph is planar or not; it does not provide any way to identify edges that should be removed to make the graph planar. Also, as was mentioned earlier, the implementation of this method tested in [12] was slower than implementations of other planarity testing algorithms.

2.2 Booth and Lueker

The next major development in planarity testing was that of the PQ-tree, created by Booth and Lueker in [2]. According to [2], the PQ-tree is a data structure designed to represent the "permutations of a set U in which various subsets of U occur consecutively." PQ-trees have two kinds of nodes: P-nodes and Q-nodes. The children of P-nodes may be permuted in any order, whereas the only acceptable permutations of the children of Q-nodes are the order in which they are listed or reverse order. The main difficulty of PQ-trees lies in the use of 11 templates which are used to determine when P-nodes and Q-nodes are added to or removed from the tree. Although the visual representations are fairly intuitive, the implementation details can be tedious.

The elements of PQ-trees are P-nodes, Q-nodes, and leaves. P-nodes and Q-nodes are interior and always have descendants, which may be other nodes or leaves. The frontier of a PQ-tree is the result of reading the leaves of a tree from left to right. The primary operation for PQ-trees is `Reduce`, which takes as input a PQ-tree, T, and a set of elements, S, which is a subset of the leaves of the PQ-tree and outputs a modified version of the original PQ-tree

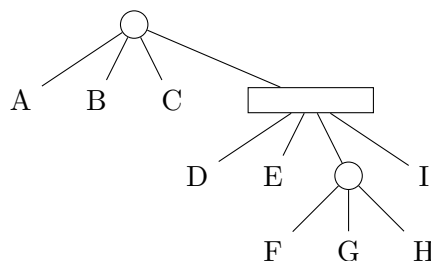in which constraints are applied so that the elements of S occur consecutively.

Figure 2.3: A PQ-tree with frontier A, B, C, D, E, F, G, H, I

To accomplish this, the PQ-tree is traversed in a separate procedure called `Bubble`, which marks leaves that are part of S as full and all other leaves as empty. A node whose children are all marked full is also marked as full, and nodes which have both full and empty children are marked as partial. After `Bubble` is complete, `Reduce` applies the templates where necessary, beginning at the bottom of the tree and moving toward the root. The goal of `Reduce` is to modify the tree so that all leaves in S occur consecutively in the frontier of the tree. To accomplish this, the templates are applied to eliminate partial nodes until at most one remains. If a partial node remains, then it must be either a P-node which contains as a child exactly one full node whose descendants are exactly the elements of S; or else a Q-node whose children are all either full or empty and whose full children occur consecutively. Notice that it is possible for `Reduce` to fail if the elements of S cannot be arranged so that they occur consecutively in the tree. In this situation, a null tree will be returned. From this point forward, the `Reduction` procedure will be defined as the procedure in which `Bubble` is performed, followed immediately by `Reduce`. `Reduction` requires as input a PQ-tree and a set S. (Note, this definition of `Reduction` is slightly different from that of [2], in which the second input to `Reduction` is a set of sets, meaning that `Bubble` and `Reduce` will be performed several times, once for each set.)

Using an iterative vertex addition approach, the PQ-tree can be used to help test a graph for planarity. This algorithm takes advantage of Theorem 1.2.1 to assume that the input graph is biconnected. As a further restriction, the vertices are assumed to be labeled from 1 to V such that vertices 1 and V are adjacent and all other vertices have at least one neighbor with a smaller label and at least one neighbor with a larger label. According to [2], it can be proven that such a labeling scheme can be created for any biconnected graph in linear time. (The interested reader should note that this labeling technique is called an s, t-numbering in the literature.)

In the PQ-tree used to test for planarity, the leaves of the tree are edges in the input graph. The first step is to construct an initial PQ-tree consisting of one P-node whose children are the edges incident to vertex 1. To insert vertex 2, reduce the tree using the set of edges whose second endpoint is 2 (easy since there will only be one, namely edge $(1, 2)$). Replace this set of edges with a P-node whose children are the edges whose first vertex is 2 (such as $(2, 3)$, etc.). Vertices 3 through $V - 1$ are inserted in the same way. In general, to insert vertex v, reduce the tree with respect to the set of edges incident to v (will include edges of the form (u, v), where $u < v$). After the reduction, all edges which have an endpoint at v will be consecutive in the frontier of the PQ-tree. Replace all of these edges with a P-node whose children are the back edges incident to v, that is, all edges of the form (v, w), where $v < w$. If the reduction operation ever fails, the graph is non-planar. Otherwise, if all vertices are successfully inserted, then the graph is planar.

The PQ-tree approach to planarity testing was the first vertex addition method that could be done in linear time. However, the disadvantage is that the numerous templates make the implementation much more complicated. This complexity provided the motivation for the development of simpler planarity testing algorithms.

2.3   Shih and Hsu

PC-trees were introduced by Shih and Hsu [15, 9] specifically to represent partial embeddings of planar graphs and were intended to be simpler to use than PQ-trees. It was later observed that PC-trees are actually a generalization of PQ-trees and that by using a free tree rather than a rooted tree all, of the PQ-tree templates could be reduced to one case. Furthermore, it was demonstrated that PC-trees can be converted to PQ-trees by adding a "dummy" node to serve as a root [10].

Despite their similarity to PQ-trees, the planarity testing algorithm which uses PC-trees is very different from that for PQ-trees. In Shih and Hsu's planarity testing algorithm, P-nodes represent cut vertices of the partial embedding, and C-nodes represent biconnected components [4]. As in PQ-trees, the children of P-nodes may be permuted in any order. Children of C-nodes are restricted to either a cyclic ordering or its reverse.

The algorithm begins by constructing a DFS tree and creating a PC-tree consisting of all of the vertices and tree arcs. Each vertex, i is visited in reverse DFS order, and all back edges incident to i and its descendants are embedded at iteration i. Essentially, at each iteration, a "terminal path" is identified, checks are made to ensure that embedding the back edges of the current iteration does not result in a non-planar graph, and C-nodes are inserted where necessary to indicate the creation and merging of biconnected components.

To explain a terminal path, the concepts of `i-subtrees` and `i*-subtrees` are introduced. At each iteration, the back edges entering vertex i are considered for embedding. The `i-subtrees` are the descendants of i which have back edges to i; `i*-subtrees` are those which have back edges to ancestors of i. Terminal nodes are identified as those which have descendants that have both `i-subtrees` and `i*-subtrees` and which do not have any other descendants that could be considered terminal nodes. A planar graph may not have

more than two terminal nodes (readers interested in a proof are referred to [15]). If there are two terminal nodes, the path between them in the PC-tree is referred to as the terminal path, and, according to [4], this is where conditions for non-planarity will occur. If there is only one terminal node, then u' is the earliest node in the tree that contains an `i*-subtree`, and the terminal path extends from the terminal node to u' in the tree.

If conditions for planarity are met (for a complete description of these conditions, refer to [3]), a C-node including the "essential nodes" on the terminal path may be inserted. The essential nodes are those which have back edges to vertices larger than i and therefore have edges that have not yet been embedded. These vertices will appear on the representative bounding cycle (RBC) of the biconnected component represented by the new C-node. In a visual representation of the graph, these vertices would appear in the cycle defining the external face. After all vertices have been examined and all back edges embedded without the conditions being met for non-planarity, the graph is declared planar.

Chapter 3

Boyer and Myrvold: Edge Addition

3.1 Overview

The planarity testing algorithm of most interest for this thesis is Boyer and Myrvold's edge addition method [5]. The algorithm "exploits the fact that subgraphs can become biconnected by adding a single edge" [5]. It essentially works by constructing several small biconnected components in a new graph, G', and merging these together when an edge is embedded whose endpoints are in what were previously separate biconnected components in G'.

Unlike other algorithms, this method does not assume that the input graph, G, is biconnected because the biconnected components will be constructed in G' as the partial embedding is created. A key observation for this to work correctly is that in a DFS tree each vertex has a path of lower numbered ancestors leading to the root. This means that all of these lower numbered ancestors must be embedded in the same face of G'. For simplicity, this face is designated as the external face of G'. As edges are embedded, a check is performed to identify vertices which have back edges in G that have not yet been

embedded, and these are marked as externally active. The externally active vertices are then required to remain on the external face of their biconnected component in G'. For this reason, a flipping operation is defined which allows the orientation of biconnected components to be flipped. For example, if the order of vertices on the cycle defining the external face were initially a, b, c, d, then the new order of vertices after the flip would be a, d, c, b. An example is given in Figure 3.1, and the pseudocode as it appears in [5] is included in Appendix A. A more in-depth explanation of the details of the algorithm will now be provided.
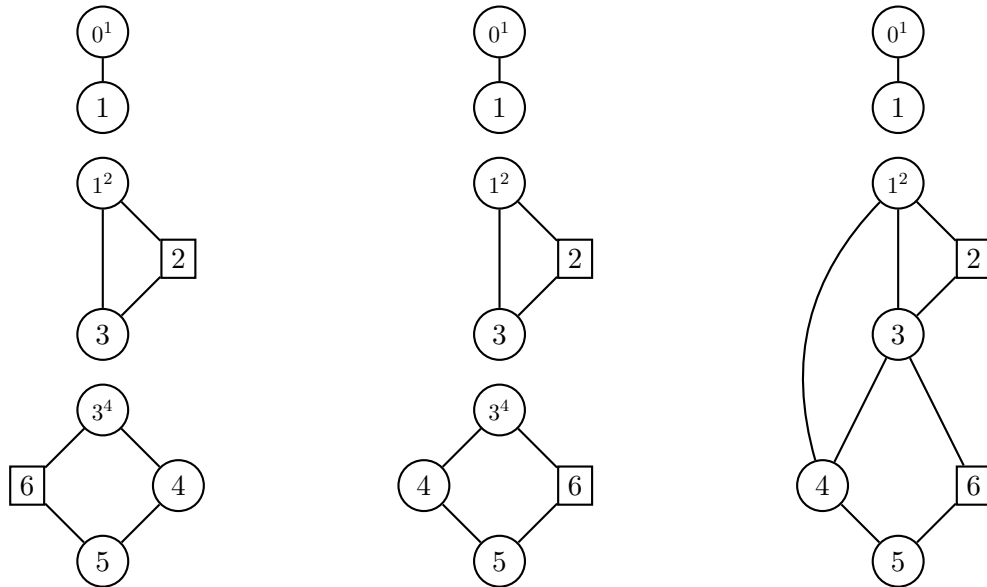
Figure 3.1: Example execution of a step in Boyer and Myrvold's edge addition algorithm, reproduced from [5]. Vertices enclosed in squares are externally active. The first part shows G' immediately before the embedding of edge $(1, 4)$. After that, the second part demonstrates the flip operation for the biconnected component with root 3^4, which must be performed because vertex 6 is externally active. The third part shows the actual embedding of edge $(1, 4)$ in such a way that the externally active vertices 2 and 6 remain on the external face.

3.2 Implementation Details

Specifically, the algorithm works in the following way. First, a DFS tree is constructed and the lowpoint values for vertices in the input graph, G are computed. Refer to Section 2.1 for information about lowpoint calculations. Also, to aid in determining quickly whether or not a vertex is externally active, each vertex, v is given a `separatedDFSChildList` which consists of the neighbors of v that are in a separate biconnected component in G'. At initialization, this list should include all neighbors of v. When a biconnected component containing v and its DFS parent is merged with a biconnected component with v as the cut vertex, the second component must contain at least one child of v, which is then removed from v's `separatedDFSChildList`.

Vertices are examined in reverse order, and all edges from a vertex to its descendants in the DFS tree are embedded before proceeding to the next vertex. Consider the embedding process for vertex v. First, create a new biconnected component in G' consisting of a "virtual vertex," v^c, vertex c, and an edge connecting these two vertices for each child c of v. Note, v may have more than one child, which would result in the creation of multiple biconnected components. Hence, the root of the new component is called v^c to distinguish it from the roots of other biconnected components. Vertex v is treated as if it is a cut vertex until an edge is added with endpoints in the biconnected components containing v and v^c, at which time vertices v and v^c are merged. After the edges between v and its children in the DFS tree are embedded, a `Walkup` of the G' is performed for all back edges incident to v and a descendant w. For convenience, call these back edges between v and its descendants the back edges of v.

3.2.1 Walkup

The goal of the `Walkup` is to identify the `pertinentRoots`, that is, the roots of the bi-connected components that will be merged as a result of embedding the back edges of v. Each vertex w with a back edge to v must be on the external boundary of its biconnected component because it is externally active. For each such vertex w, begin with w in G' and proceed simultaneously clockwise and counterclockwise, visiting vertices on the external face in w's biconnected component until the root, say r^s is found. Vertex r^s is added to the list of `pertinentRoots`, and then the biconnected component containing r is searched for its root vertex and so on until r^s is equal to v^c, where c is a child of v in the DFS tree.

3.2.2 Walkdown

After the `Walkup`, a `Walkdown` of G' is performed for each DFS tree child c of v. The overall aim of the `Walkdown` is to actually embed the back edges incident to v and each of its descendants. If a descendant, w, is in a separate biconnected component in G', then two or more biconnected components may need to be merged. Essentially, the `Walkdown` begins at vertex v^c and must explore paths in both directions (clockwise and counterclockwise) along the external face cycle. When a vertex in `pertinentRoots` is found, say r, the search continues in the biconnected component rooted at r^s. The `Walkdown` is searching for a path along which edge (v, w) may be embedded; therefore, it cannot traverse past an externally active vertex because embedding the edge there would "block in" that vertex, making it impossible to embed a future edge. When the `Walkdown` descends to r^s, the traversal of this biconnected component will either proceed in the same direction as that of the previous biconnected component or else in the opposite direction, depending whether or not there are externally active vertices present. If the traversal proceeds in the opposite direction,

the component must be flipped before the edge is embedded.

After the `Walkdown`, it is important to ensure that all back edges incident to v and its descendants were successfully embedded. If not, the graph is known to be non-planar, and a subgraph homeomorphic to either K_5 or $K_{3,3}$ is identified.

After all vertices have been examined and all edges are embedded, a planar embedding indicating the order of edges around each vertex and the orientation of the biconnected components in G' can be retrieved. Refer to [5] for information on how to do this.

Chapter 4

Parallelizing Edge Addition

4.1 Shared Memory Model

To parallelize Boyer and Myrvold's edge addition algorithm, a shared memory model was selected. In the course of execution, the algorithm constructs several small biconnected components. If a message passing model were employed, about half of these biconnected components would have to be sent between processes at every merge step, which could be prohibitively expensive, especially for large graph sizes.

4.2 Initialization

Before beginning, a DFS tree should be constructed to assign labels to vertices, and edges should be sorted according to the following constraints:

- For each edge $(v, w), v < w$

- Edges are sorted by descending order of v and then by ascending order of w, e.g., $(2, 3), (1, 2), (1, 3)$

Edges are to be distributed equally among the threads, so it is important that each edge appear only once. Recall from Chapter 3 that a back edge from vertex v to a descendant w is embedded when v is explored. If a list were made of the order in which edges were embedded in the sequential algorithm, edge $(5, 6)$ would be added before $(3, 5)$. Recall that when a vertex v is visited in the sequential algorithm, the edges from v to its children in the DFS tree are embedded first, followed by any other back edges from v to its descendants. Further, recall that in the DFS tree every vertex has a smaller number than any of its descendants. Thus, as long as edges from v to its descendants are embedded by ascending order of the descendants of v, the every edge from v to one of its children is guaranteed to be embedded before the edges from v to the descendants of that child. The sorted order of the edges then roughly mimics the order in which edges are embedded by Boyer and Myrvold's algorithm.

Throughout, assume that there are t threads numbered from 1 to t. Edges will be distributed evenly among threads in reverse sorted order, so thread t will be given the edges at the beginning of the sorted list of edges. The input graph G and the DFS tree T created when labeling the vertices are assumed to be accessible by all threads. Each thread will have its own G', and as a notational convenience to distinguish between the G' of different threads, G_i will be used to refer to the G' constructed by thread i.

4.3 General Procedure

The general idea of the parallelized version of the algorithm is that each thread is assigned a portion of the edges, which are embedded into G_i in a similar fashion to that of the sequential algorithm, with some exceptions that will be discussed later. After each thread has finished, the G_i are merged pairwise (if there are an odd number of threads, thread 1

does not merge). Thus, all of the G_i will be merged in $\log_2(t)$, steps where t is the number of threads. As a notational convenience, renumber the threads after each merge step, so that if there are t threads before the merge, then there are $\lceil t/2 \rceil$ threads numbered sequentially beginning with 1.

The exceptions alluded to earlier will now be explained. First, observe that G_i is initially empty, so embedding some edges can result in the creation of new vertices in G_i. However, because the edges are distributed among threads, it could be the case that if all edges were embedded a vertex, say v, might be created in several different G_i. If multiple copies of the same vertex are created, these will eventually have to be merged, which would require locating all of the copies and then identifying sets of conditions in which vertices may and may not be merged. Although virtual vertices may at first appear to do exactly this, they are slightly different in that they are created in a controlled way; i.e., they are only created between parents and children in the DFS tree. Rather than creating multiple copies of the same vertex and attempting to merge them later, edges which could cause this problem are marked as pending and temporarily skipped until a merging operation occurs. An edge (v, w) in thread i should be marked as pending when both of the following conditions hold:

- w is not a child of v in the DFS tree

- v or w is not already present in G_i

This means that immediate children of v in the DFS tree are always embedded immediately. Any edges that could cause two biconnected components in G_i to merge may also be embedded. Edges which could result in merging a component in G_i with one in another thread are marked as pending. Another attempt to embed them will be made when threads are merged. Note that because of the constraints defined in Section 4.1, w must be a descendant of v in the DFS tree. The merge operation occurs after all edges except those

marked as pending have been embedded. Because edges are distributed equally among threads, it is possible that edges from a vertex v to its descendants are divided among multiple threads. In the case where v has one DFS child, this edge will be the first one embedded, edges to other descendants will be embedded after the merge operation. If v has multiple children in the DFS tree, the edges to these children may or may not be on the same thread, but they are embedded first, regardless.

Suppose that during the merge operation, G_i is merged with G_{i+1}. All of the biconnected components in G_{i+1} are added to G_i, and the pending edges marked by G_i are examined to determine if any of them may now be embedded.

Theorem 4.3.1. *Pending edges marked in thread $i + 1$ will remain pending after a merge operation between G_i and G_{i+1}.*

Proof. Consider one such edge (v, w) that was marked as pending in thread $i + 1$. It is already known that w was not present in G_{i+1}, so the edge can only be embedded in G_i if w was present in G_i before the merge occurred. Consider how it would be possible for vertex w to be in G_i already. It must be the case that an edge with an endpoint at w was successfully embedded, i.e., it was not marked as pending. Also, recall that edges are distributed among threads in reverse order. Clearly, for every edge (t, u) in G_i, $t \leq v$. Therefore, any edge in thread i incident to w must be a back edge from w to an earlier numbered vertex, say s, where $s < v$. In order for edge (s, w) to be embedded, it must be the case that at least one of the conditions listed above is violated. Since the goal is to determine how it was possible for vertex w to be added to G_i in the first place, assume that the second condition holds; therefore, it must be the case that w is a child of s in the DFS tree. However, it has already been stated that w has a back edge to v, so v must be an ancestor of w in the DFS tree. Since s is the parent of w, it must also be the case then that v is an ancestor of s in

the DFS tree. However, if v is an ancestor of s, then $v < s$, a contradiction to the earlier statement that $s < v$. Thus, w could not have been present in G_i before the merge, and (v, w) will not be embedded in G_i after the merge occurs. $\qquad\square$

4.4   Revised Embedding Process

Boyer and Myrvold's algorithm relies on the fact that each vertex has a path through lower-numbered vertices to the DFS tree root, which means that at any point in the execution, all as yet unprocessed vertices can be embedded in a single face of G'. This face is assumed to be the external face. Due to the fact that edges are not necessarily processed in this order, this constraint no longer holds, so some additional observations are necessary to allow the parallel algorithm to work.

First, we observe that the criteria for externally active vertices still hold, despite the parallel nature of the algorithm. On any given thread, edges are processed by decreasing value of v, so it is still the case that every vertex has a path in the DFS tree through lower numbered, unprocessed vertices, and all of these must still be embedded in a single face of G', which will be assumed to be the external face.

In addition to the externally active vertices, consider an edge (v, w) in thread i which is marked as pending, where vertex v is in G_i. The problem is that v can not simply be assumed to be on the external face of G_i. In fact, it is fairly easy to construct an example in which this assumption causes a planar graph to be called non-planar. Suppose that all vertices with unembedded back edges were marked as externally active and kept on the external boundary. Consider the graph in Figure 4.1, which is clearly planar.

Now, imagine that the edges are distributed in such a way that the biconnected components depicted in Figure 4.2 are created in two separate threads. All vertices in this figure
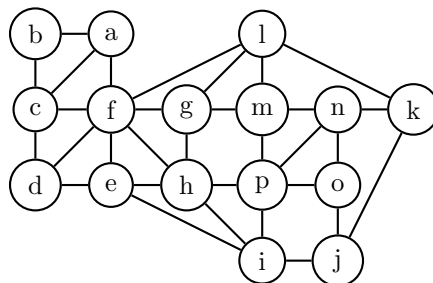
Figure 4.1: Graph to be used in a counter-example demonstrating that vertices cannot all be assumed to be externally active in parallelizing edge addition.

would be marked as externally active because they all have unembedded edges. Those in the subgraph on the left all have unembedded edges to the vertices in the subgraph on the right and vice versa. To indicate that the vertices are externally active they are drawn in squares.
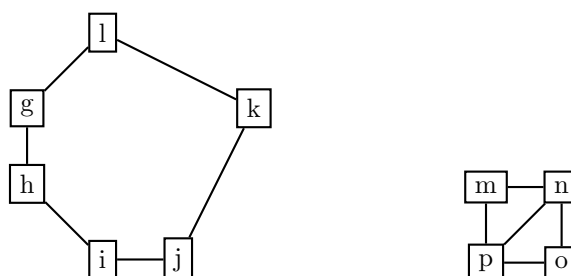

Figure 4.2: Two potential biconnected components created from the graph in Figure 4.1 in separate threads

Figure 4.3 demonstrates what would happen if these threads merged their results as they would in the original, sequential algorithm and began embedding edges that had previously been marked as pending. The component on the right in Figure 4.2 would have to be flipped first; otherwise edge (j, o) could not be embedded. After that, embedding edge (h, p) is still an issue because inserting it in one orientation results in blocking externally active vertex

i, and inserting it the other direction blocks vertices g and l. This failure to embed edge (h, p) would result in the graph being declared non-planar, despite the fact that it clearly is planar.
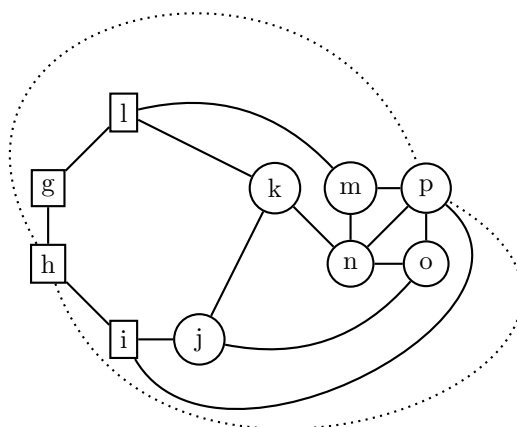

Figure 4.3: Problems arise when attempting to embed edge (h, p) because externally active vertices are blocked.

Clearly, the solution to this problem revolves around realizing that the component on the right side of Figure 4.2 should be embedded *within* the component on the left side of the same figure. This leads to the following definitions and theorem.

Definition 4.4.1. *If there is an edge (v, w) in thread i which is marked as pending, then v is called a partially active vertex.*

Definition 4.4.2. *The process of embedding one biconnected component within another is called submerging.*

Theorem 4.4.1. *If a graph is planar, then for any biconnected component b_i in thread i, which must be merged with biconnected component b_j in thread $i + 1$ there exists a set of partial vertices in b_i which if edges were added between them would form a cycle within which b_j could be submerged.*

For an example of Theorem 4.4.1, consider Figure 4.3. The component referred to as b_i is the one consisting of vertices g, h, i, j, k, l, and component b_j includes vertices m, n, o, p. The set of partial vertices in b_i includes all of the vertices because they all have back edges to vertices in b_j. Note that the vertices in b_j are still considered externally active in the sense that they have unembedded back edges to lower numbered vertices. Although the submerging process can result in these vertices no longer being on the external face of their new biconnected, this is not a problem because all of their back edges are to vertices on the cycle into which their component has been submerged.

Figure 4.4 shows the two cases for what a final graph could look like after the biconnected component consisting of vertices v, w, x, y, z, call this component B, has been submerged. It is either the case that no elements of B will remain on the external face or that some of them will be on the external face. Vertices v, y, z are drawn in squares to indicate that they were externally active before B was submerged because they had unembedded back edges. The graph on the left represents the case in which B is completely submerged, and no vertices in B are on the external face. Cycle C consists of vertices a, b, c, d, e, f. Note that vertices v, y, z are no longer externally active after the embedding: it is assumed that their only back edges were to members of C. The graph on the right shows the case in which B is not completely submerged in that some vertices remain on the external face. Cycle C consists of vertices a, e, f, and again note that vertices v, y, z are drawn in squares to indicate that they were externally active before the submerging process. Although vertices
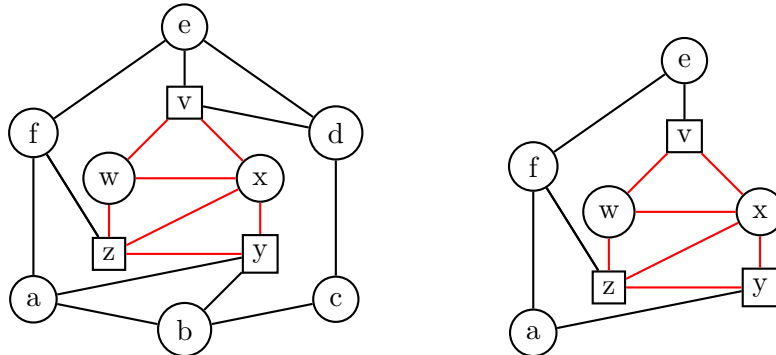
Figure 4.4: Graphs demonstrating the two cases for submerging the biconnected component consisting of vertices v, w, x, y, z.

v and y may still be externally active, vertex z is assumed to have all back edges embedded now.

Proof of Theorem 4.4.1. Recall that when the edges of thread $i+1$ are embedded, all externally active vertices remain on the external boundary. Clearly, the biconnected component b_j cannot be embedded across multiple faces of b_i, as this would cause at least one pair of edges to intersect. Therefore, b_j must be embedded in a single face of b_i, which may or may not be the external face, if it may be embedded at all. Every face in the graph is defined by a cycle of vertices. Call the cycle around the face in which b_j is embedded C. If b_j is embedded on the external face of b_i, then C consists of the vertices that were on the external face cycle of b_i before b_j was embedded. Every vertex to which the externally active vertices in b_j have a back edge must be in cycle C. Therefore, every partial vertex in b_i that is adjacent to an externally active vertex in b_j must be in C.

It is possible that C includes additional vertices that are not partial vertices; however, these do not have back edges to any vertices in b_j (otherwise, they would be partial vertices), and, if they have back edges to vertices in C other than their two immediate neighbors in

the cycle, these may be embedded "outside" of cycle C, otherwise some vertices would not have been included in C.

Consider two partial vertices in C, say v_1 and v_2 which are separated in C (in one direction) by one or more non-partial vertices, but no partial vertices. For the theorem to hold, it must be possible to embed an edge between v_1 and v_2 within cycle C, eliminating the non-partial vertices between v_1 and v_2 from C without causing any edges to cross. As an example, consider the graph on the left side of Figure 4.4. The two partial vertices b and d are separated by a non-partial vertex c. For the theorem to hold, it must be possible to insert an edge (b, d) into the graph if the edge does not already exist. Note that if such an edge had already been embedded, the theorem would hold trivially. If this edge were inserted, it would not intersect any outgoing edges from the non-partial vertices for the following reasons. If a non-partial vertex had outgoing edges to other vertices in the cycle (other than than its two immediate neighbors in the cycle), these edges could not span across any partial vertices because they would intersect with edges between the partial vertices and b_j. In Figure 4.4, vertex c could not have edges to vertices on the cycle other than b and d that were embedded within cycle C because these edges would intersect with edges incident to b or d and the submerged component. Furthermore, since c is non-partial, it does not have any back edges to vertices in the submerged component. Therefore, edge (b, d) could not intersect any edges incident to the non-partial vertex c.

The edge between v_1 and v_2 cannot intersect any edges between partial vertices in the cycle either because there are, by assumption, no partial vertices in the cycle between v_1 and v_2. In Figure 4.4, it is fairly easy to see that no edge between b and any other partial vertex could intersect with an edge from b to d.

The last case is that edge (v_1, v_2) intersects with an edge between a partial vertex in C and b_j, but this is also impossible because of the assumption that there are no partial

vertices on the cycle between v_1 and v_2. In Figure 4.4, the edge (b, d) could not intersect an edge between any other partial vertex and the submerged component. Since vertex c is non-partial, it cannot have any edges to vertices in the submerged component. Therefore, the theorem holds. □

4.5   Summary of Method

In order to account for the possibility that some biconnected components must be submerged, certain changes must be made to the algorithm. Below is a brief overview of how the parallelized algorithm will work:

1. Perform a DFS to label all vertices and construct a DFS tree.

2. Divide the edges evenly between threads as described in Section 4.2.

3. For the first iteration, embed edges as described in [5], with the exception of pending edges as described in Section 4.3.

4. Merge threads pairwise. If there are an odd number of threads, thread 1 is left out. For each pair of threads i and $i + 1$, add the biconnected components of G_{i+1} to G_i.

5. For each edge (v, w) that was marked as pending by thread i, determine if it may now be embedded. If not, leave it marked as pending. Otherwise, identify the biconnected components of G_{i+1} that should be submerged and submerge them. If one or more components cannot be submerged, mark the graph as non-planar.

6. Continue until all edges that were pending in thread i have been reconsidered for embedding. Some edges may still be pending.

7. Repeat steps 4 through 6, merging threads pairwise until the graph is found to be non-planar or only one thread remains and all edges have been embedded.

In order for this algorithm to work effectively, some additional data and procedures must be defined. A description of some of the new implementation details will now be provided.

4.6   Implementation Details

In order to accommodate the changes introduced in the parallelization, some additional information and procedures need to be included. The goal is to keep track of the cycle of vertices into which biconnected components will be embedded in the future. When a partial vertex is identified, it is added to a list of partial vertices local to the current thread. The first time this occurs, create a new biconnected component consisting of the partial vertex p and a temporary vertex B, which represents the biconnected component that will be submerged at some time in the future. Each time a partial vertex is identified, embed an edge from the newest partial vertex to B. The idea is that these edges represent the back edges between partial vertices and the biconnected component that will be embedded in the future. It also ensures that a cycle containing all of the partial vertices is maintained. Note that B need not remain in the external face, since it represents a component that will be submerged. In fact, B should not be embedded in the external face if it would obstruct externally active vertices; however B should be kept on the external face whenever possible. Any partially active vertices encountered in the future must also have an edge to B. This maintains the cycle and ensures that if an edge to B cannot be embedded then the graph will be declared non-planar. Figure 4.5 demonstrates this idea by showing where vertex q would be inserted in place of the biconnected component formed by vertices m, n, o, p.
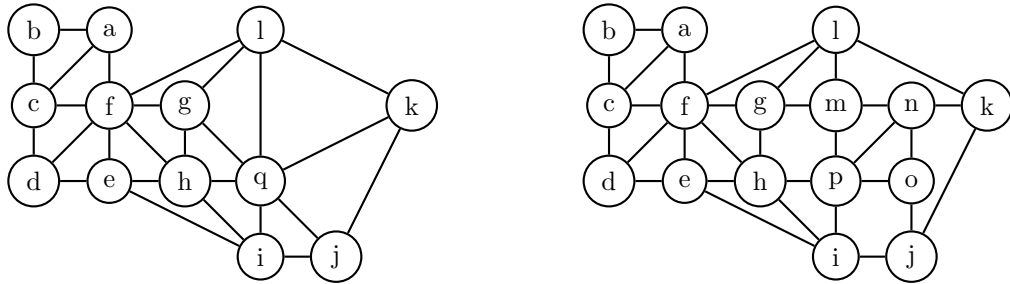
Figure 4.5: Vertex q is inserted as a temporary placeholder for the biconnected component consisting of vertices m, n, o, p until that component may be submerged.

4.6.1 A Special Case

It is possible that a biconnected component, C, which itself contains partially active vertices will need to be submerged within another component D. There are two cases for this situation:

1. The temporary vertex in C is on the external face.

2. The temporary vertex in C is not on the external face.

Call the temporary vertices in C and D V and W, respectively. In case 1, V and W will essentially merge so that all vertices in C which had edges to V will now have edges to W, and V will no longer exist. Pending edges between vertices in C and D will then be embedded.

If instead case 2 applies and the V is not on the external face of C, then there are two possibilities:

1. All pending back edges incident to W and other vertices in D may be embedded as a result of submerging C.

2. Some back edges incident to W and other vertices in D are still pending after C is submerged.

In the first case, W may simply be eliminated and all edges that were incident to W may be embedded. For the second case, recall that it was earlier stipulated that the temporary vertex be kept on the external face whenever possible, which means that V is embedded within a cycle. If not all back edges incident to W and vertices in D are embedded, this implies that some of the partial vertices in D are incident to elements of the biconnected component represented by temporary vertex V. If this is the case, then the graph must be non-planar. This is because the component represented by V must be submerged in the interior of C (otherwise V would be on the external face), and yet some vertices in D are adjacent to vertices in the biconnected component represented by V (otherwise all edges incident to W could have been embedded).

4.6.2   Submerging Components

After components which must be submerged are identified, the only task remaining is to actually submerge them. This procedure is similar to the original embedding process. When threads i and $i+1$ are merged, the edges of i that were marked as pending are reconsidered for embedding. Edges (v, w) that are no longer pending are embedded by descending order of v and ascending order of w (i.e., the same order in which edges were sorted prior to distribution among threads). Before embedding, identify the partial vertices that will no longer be partial after all pending edges are embedded, and remove the edges between these partial vertices and their temporary vertex. This is important to prevent incorrect identification of the graph as non-planar. If the temporary vertex has no edges left, delete it from the graph.

Now, consider the first edge, (v, w) that is to be embedded. Clearly, v and w are in separate biconnected components, one of which was on thread i and the other of which was previously on thread $i+1$ (otherwise the edge and both vertices would have been on thread i, and the edge would not have been marked as pending). The biconnected component containing w will be inserted in the same place that the temporary vertex of the biconnected component containing v was. The first pending edge examined may be inserted without difficulty. To embed subsequent pending edges, care must be taken that edges are not permitted to cross and that "externally active" vertices in the component being submerged remain on that component's external face. Note that because the component has already been submerged, these externally active vertices may not actually be on the external face of their new biconnected component. This may be accomplished using modified versions of `Walkup` and `Walkdown`. The change that must be made is that if the component must be flipped, then the pending edges that have been embedded must be rechecked to ensure that flipping the component will not result in any of them being incorrectly embedded. One fairly straightforward way to do this would be to keep a stack containing edges that are embedded during the submerging of a biconnected component. If the component is successfully embedded without need to flip, then the stack is cleared after all edges are inserted. Otherwise, if a flip occurs, pop the stack to identify edges which must be reconsidered for embedding. Delete edges from the graph as they are popped from the stack, and then begin again trying to reinsert them in order. If they cannot be successfully embedded, declare the graph non-planar; otherwise the graph is planar.

Chapter 5

Conclusions and Future Work

This thesis presents a new parallel planarity testing algorithm based on Boyer and Myrvold's sequential edge addition algorithm and an argument for its correctness. Although a complete analysis of the complexity has not been conducted, this is one area for future research. Another potential area for research is implementation. This algorithm has not yet been implemented. Bader and Sreshta [1] state that they "have efficient shared memory implementations for most of the major steps involved in the[ir] algorithm," which seems to imply that they do not have a complete implementation. A search for implementations of other parallelized planarity testing algorithms was also unsuccessful; nor do there seem to be any published articles addressing this issue. Testing is needed to determine how much of a speed advantage these different parallelization techniques offer. It may be that they offer a significant speedup, or it could be that none of the proposed solutions offers any sizeable advantage.

Bibliography

[1] David A. Bader and Sukanya Sreshta. A new parallel algorithm for planarity testing. Technical Report UNM-ECE Technical Report 03-002, University of New Mexico, October 2003.

[2] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[3] J.M. Boyer. Additional pc-tree planarity conditions. In *Proceedings of the 12th International Conference on Graph Drawing 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 82–88, 2005.

[4] J.M. Boyer, C.G. Fernandes, A. Noma, and Jr. J.C. de Pina. Correcting and implementing the pc-tree planarity algorithm. `http://citeseer.ist.psu.edu/62669.html`, 2003.

[5] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA, 1979.

[7] Bernhard Haeupler and Robert E. Tarjan. Planarity algorithms via pq-trees (extended abstract). *Electronic Notes in Discrete Mathematics*, 31:143–149, 2008.

[8] John Hopcroft and Robert Tarjan. Efficient planarity testing. Technical Report TR 73-165, Cornell University, April 1973.

[9] Wen-Lian Hsu. An efficient implementation of the PC-Tree algorithm of shih & hsu's planarity test. Technical report, Institute of Information Science, Academia Sinica, 2003.

[10] Wen-Lian Hsu and Ross M. McConnell. *Handbook of Data Structures and Applications*, volume 4 of *Computer & Information Science Series*, chapter PQ Trees, PC Trees, and Planar Graphs. Chapman & Hall/CRC, 2004.

[11] Philip N. Klein and John H. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Sciences*, 37:190–246, 1988.

[12] Giuseppe Liotta, editor. *Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*. Springer, 2004.

[13] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*, volume 32 of *Annals of Discrete Mathematics*. Elsevier, 1988.

[14] Vijaya Ramachandran and John Reif. Planarity testing in parallel. *Journal of Computer and Systems Sciences*, 49:517–561, 1994.

[15] Wei-Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theoretical Computer Science*, 223:171–191, 1999.

Appendix A

Boyer and Myrvold: Edge Addition

The pseudocode for Boyer and Myrvold's edge addition algorithm as it appears in [5] is reproduced here for convenience.

PLANARITY(G)

```
 1   Perform depth first search and lowpoint calculations for G
 2   Create and initialize G' based on G, includng creation of
     separatedDFSChildList for each vertex, sorted by child lowpoint
 3   for each vertex v from V − 1 down to 0
 4       do for each DFS child c of v in G
 5               do Embed tree edge (v^c, c) as a biconnected component in G'
 6           for each back edge of G incident to v and a descendant w
 7               do WALKUP(G', v, w)
 8           for each DFS child c of v in G
 9               do WALKDOWN(G', v^c)
10           for each back edge of G incident to v and a descendant w
11               do if (v^c, w) ∉ G'
12                   ISOLATEKURATOWSKISUBGRAPH(G', G, v)
13                   return (NONPLANAR, G')
14   RECOVERPLANAREMBEDDING(G')
15   return (PLANAR, G')
```