8-2015

The PC-Tree algorithm, Kuratowski subdivisions, and the torus.

Charles J. Suer
University of Louisville

THE PC-TREE ALGORITHM, KURATOWSKI SUBDIVISIONS, AND THE TORUS

By

Charles J. Suer
B.S., University of Dayton, 2010
M.A., University of Louisville, 2013

A Dissertation
Submitted to the Faculty of the
College of Arts and Sciences of the University of Louisville
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy in Applied and Industrial Mathematics

Department of Mathematics
University of Louisville
Louisville, Kentucky

August 2015

THE PC-TREE ALGORITHM, KURATOWSKI SUBDIVISIONS, AND THE TORUS

By

Charles J. Suer
B.S., University of Dayton, 2010
M.A., University of Louisville, 2013

A Dissertation Approved on

13 May 2015

by the following Dissertation Committee:

Dr. André Kézdy, Dissertation Director

Dr. Csaba Biró

Dr. D. Jake Wildstrom

Dr. Robert Powers

Dr. Ki-Hwan Bae

To Erin.

ACKNOWLEDGMENTS

Thank you to my wife, parents, siblings, and in-laws who have loved and supported me through this endeavor.

To Csaba Biró for introducing me to other areas of Combinatorics.

To Thomas Riedel, Robert Powers, Jake Wildstrom, Adam Jobson, and all of the faculty at the University of Louisville for a wonderful graduate school experience.

To Erika Foreman, Allison Perkins, Chad Money, Rasitha Jayesekere, Heather Hunt, Lucas Hoots, Carlos Paniagua, and all of my classmates for providing opportunities to laugh, complain, and succeed together.

Most importantly, thank you to my advisor, André Kézdy for all of his mathematical, professional, and personal mentoring.

ABSTRACT

THE PC-TREE ALGORITHM, KURATOWSKI SUBDIVISIONS, AND THE TORUS

Charles J. Suer

13 May 2015

The PC-Tree algorithm of Shih and Hsu (1999) is a practical linear-time planarity algorithm that provides a plane embedding of the given graph if it is planar and a Kuratowski subdivision otherwise. Remarkably, there is no known linear-time algorithm for embedding graphs on the torus. We extend the PC-Tree algorithm to a practical, linear-time toroidality test for $K_{3,3}$-free graphs called the PCK-Tree algorithm. We also prove that it is NP-complete to decide whether the edges of a graph can be covered with two Kuratowski subdivisions. This greatly reduces the possibility of a polynomial-time toroidality testing algorithm based solely on edge-coverings by subdivisions of Kuratowski subgraphs.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This dissertation focuses on the problem of efficiently (polynomial-time) and practically recognizing which graphs can be embedded on the torus. To design an algorithm for this problem, we adapt a planarity checking algorithm based on PC-Trees. This chapter introduces the necessary background on surfaces, graph theory, embeddings, and algorithms. An introduction to the planarity checking algorithm based on PC-Trees is more involved and is presented in Chapter 2.

1.1 – Surfaces

The plane is the orientable surface with the smallest genus. The *genus* of a surface is the largest number of non-intersecting simple closed curves that can be drawn on the surface without separating it. The genus of the plane is 0 since any simple closed curve separates the plane into two regions by the Jordan Curve Theorem [16]. Simply an infinite flat surface, the plane is the setting for a multitude of problems in all of mathematics, not just graph theory. Another orientable surface is the sphere. While topologically different from the plane, the sphere has the same genus as the plane and the two surfaces are equivalent in the sense that a finite graph is embeddable on the plane if and only if it is embeddable on a sphere. The torus is the orientable surface of genus 1 and is not equivalent to the sphere. While the torus is often viewed as a teacup (essentially a sphere with a "handle"), this paper will use the description of it as the surface of a donut.

In order to represent the torus on a planar surface such as the sheet of paper (or computer screen) on which this document is printed, a series of simple cuts are made. With some understanding of convention, the torus can be fully represented. Make a cut through the

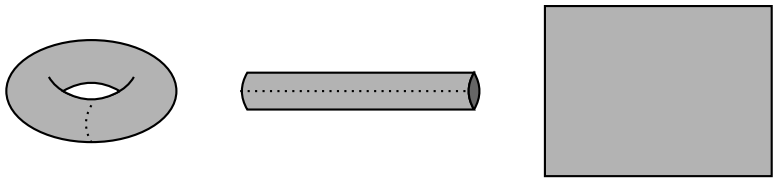

Figure 1.1: The cuts needed to represent the torus as a flat rectangle.

torus from the hole to the outside, then un-bend the donut to form a cylindrical tube. Make another cut along the length of the cylinder and unroll the surface into a rectangle. See figure 1.1. The left and right sides of the rectangle represent the first cut made, and the top and bottom sides represent the second cut made. Identify the left and right sides as well as the top and bottom. In particular, this means that when an edge of a graph reaches the left side, it reappears on the right side at the same "height." The same applies to the top and bottom of the rectangle.

1.2 – Graph Theory Basics and Types of Graphs

This dissertation will use standard graph theory terminology and notation [7]. In particular, the notation K_n represents the complete graph on n vertices and $K_{m,n}$ represents the complete bipartite graph on partite sets of size m and n. Many of the graphs dealt with in this paper are in these two classes. All graphs in this paper are assumed to be finite, simple, and undirected unless stated otherwise.

There are three edge operations that are essential to this paper. *Deleting* the edge e from the graph G is the process of simply removing the edge e from the edge set. This is represented by $G - e$ and results in a graph with the same vertex set and one fewer edge than G. *Contracting* the edge e between vertices u and v in the graph G is the process of deleting the edge e and identifying the endpoints u and v. This is represented by G/e and results in a graph with one fewer vertex and at least one fewer edge than G (any multiple edges are removed to keep the graph simple). Finally, *subdividing* an edge e between vertices u and v in the graph G is the process of replacing e with a new vertex w and the edges uw and wv. This results in a graph with one more vertex and one more edge than G. See Figure 1.2.

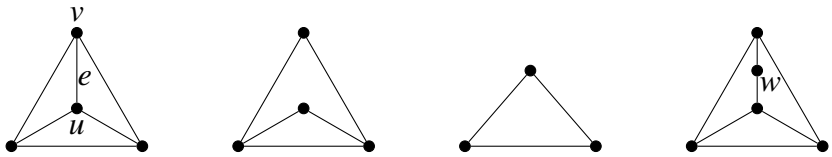

Figure 1.2: A graph with edge e and the results $G - e$, G/e, and subdividing e.

A graph G is a *subdivision* of a graph H if G can be obtained from H by successive subdivisions of edges. In other words, if G can be obtained by replacing some edges of H with disjoint paths, then G is a subdivision of H. G is a *TH* or an *H-subdivision* and H is a *homeomorph* of G and a *topological minor* of any graph containing G as a subgraph. Finally, H is a *minor* of G if a graph isomorphic to H can be obtained from a subgraph of G by contracting some edges. Denote this $H \prec_m G$ and G is an *IH*. See Figure 1.3.

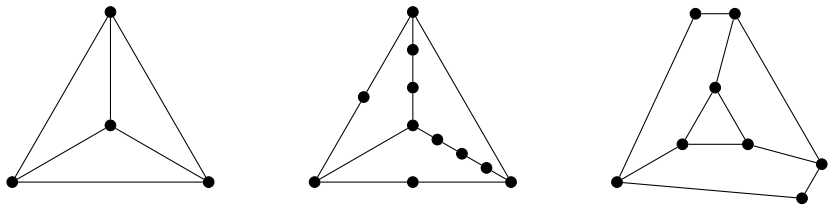

Figure 1.3: A graph G, with examples of a *TG* and an *IG*.

1.3 – Embeddings

Intuitively, a drawing of a graph on a surface produces an *embedding*. Generally, the most useful and interesting embeddings are those in which no edges cross (except at their endpoints). A graph is *embeddable* on a surface if there is an embedding of the graph on that surface with no crossing edges. The regions formed by the vertices and edges of the graph in such an embedding are called *faces*. If a graph does not have an embedding on a surface with no edges crossing, then the graph is *not embeddable* on that surface.

Planar and Toroidal Graphs

A *planar* graph is a graph that is embeddable on the plane. Planar graphs are used in a multitude of applications such as circuit design and network design. It is very easy to see that any planar graph is also embeddable on the sphere. It is only slightly harder to see that

the converse is also true. This can be rigorously proven using stereographic projections, but conceptually one can imagine that the sphere is made of very stretchy rubber and that a hole punctured in one of the faces of the embedded graph can be expanded until the sphere lies flat and the graph is thus embedded on the plane. Graphs that are embeddable on the plane have been characterized in many ways and have fast detection algorithms. The orientable surface of next highest genus has been less well studied and understood. This leads us to the study of the torus.

A *toroidal* graph is a graph that is embeddable on the torus. Toroidal graphs also have many applications, although they are more complex than planar graphs. They are used in network design, as many computer networks have a toroidal structure, which provides some stability in the way a planar network would, but with added flexibility.

For an embedding of a graph on a surface, the *rotation system* consists of a cyclic list for each vertex. This cyclic list contains the neighbors of that vertex in the clockwise order in which they appear in the embedding. Given a rotation system, it is easy to construct the embedding and vice versa.

Euler's Formula for Genus g

Euler's Formula states that in any planar embedding of any graph, the number of vertices minus the number of edges plus the number of faces is always equal to 2. However, there is a stronger result that applies to all orientable surfaces. An embedding of a graph on a surface is a *2-cell* embedding if every face of the embedding is homeomorphic to an open disk. Every planar embedding is a 2-cell embedding.

Theorem 1.1. If a graph G with n vertices, and e edges has a 2-cell embedding with f faces on an orientable surface with genus g, then

$$n - e + f = 2 - 2g.$$

The value $2-2g$ is known as the *Euler characteristic* of the surface. In particular, the Euler characteristic of the plane (genus 0) is 2 and the Euler characteristic of the torus (genus 1) is 0.

Kuratowski's and Wagner's Theorems

Planar graphs have been widely studied for many years and as a result are very well understood. There are many theorems that characterize planar graphs, but this paper will focus on two of the most important. First is Kuratowski's Theorem, proved in 1930 [19].

Theorem 1.2. A graph G is planar if and only if it does not contain a subdivision of K_5 or $K_{3,3}$.

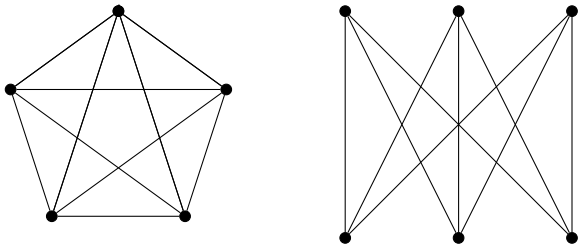

Figure 1.4: The Kuratowski graphs K_5 and $K_{3,3}$.

Subdivisions of these two graphs are often referred to as *Kuratowski* graphs. See figure 1.4. In 1937, a similar theorem appeared [27]. Wagner's Theorem also references the Kuratowski graphs, but in a slightly different manner.

Theorem 1.3. A graph G is planar if and only if K_5 and $K_{3,3}$ are not minors of G.

It may seem that Kuratowski's Theorem is stronger since a subdivision can be viewed as a minor, but the two results are actually equivalent. However, as detailed in the next subsection, the distinction between minors and subdivisions becomes important when the genus of the surface increases.

Topological and Minor-Minimal Obstructions

Two non-planar graphs have already been mentioned and non-toroidal graphs are discussed in Chapter 3. A graph G that is not embeddable on a surface S such that deleting any edge of G results in a graph that is embeddable on S is a *topological obstruction* for the surface S. Similarly, if deleting or contracting any edge of G results in a graph that is embeddable on S, then G is a *minor-minimal obstruction* for S. Every graph that is not embeddable on S contains a subdivision of a topological obstruction. Similarly, every such

graph must contain a minor-minimal obstruction as a minor. Every minor-minimal obstruction is a topological obstruction (for a surface S), but not necessarily vice-versa.

Kuratowski's Theorem identifies the topological obstructions for the plane as K_5 and $K_{3,3}$. Wagner's Theorem happens to identify those same graphs as the only minor-minimal obstructions for the plane. This is simply coincidence, and there is no guarantee that the two sets are the same in general. For example, there are 35 minor-minimal obstructions for the projective plane but 103 topological obstructions. This result is due to Glover, Huneke, and Wang who identified the obstructions [12] and to Archdeacon who verified that there are no additional obstructions [1].

No theorem analogous to Kuratowski's or Wagner's has been proven for the torus. However, several thousand obstructions of both types have been identified: 239,322 topological obstructions and 16,629 minor-minimal obstructions [6], [10], [17], [20], [21], [29].

The Robertson-Seymour Theorem

In 1990, Robertson and Seymour published the eighth paper in the Graph Minors series [23] in which they prove Wagner's Conjecture (stated to be an unpublished conjecture in [22]):

Theorem 1.4. The set of topological obstructions for any surface is finite.

Theorem 1.4 implies that there exists a theorem like Kuratowski's for every surface. Since the set of minor-minimal obstructions is a subset of the set of topological obstructions, it must also be finite. This means there is also a theorem like Wagner's for every surface. Because their proof is non-constructive, the Robertson-Seymour Theorem does not provide the sets of obstructions.

1.4 – Algorithms

A *graph algorithm* is a set of instructions used to determine information about a given graph, or to perform some operation on a given graph. These are often implemented for use by a computer. In particular, algorithms that determine if a graph is embeddable on a

surface are of interest in this dissertation. Before discussing these in more detail, the concept of algorithm complexity must be introduced.

Complexity

The worst-case, *time-complexity* (or just *complexity*) of an algorithm is a measure of how quickly the algorithm can perform its function. It is usually given as the maximum number of steps the algorithm must perform in the worst case, often in terms of the number of vertices in the graph. Some simple algorithms perform a number of steps that does not depend on the number of vertices. For instance, an algorithm that deletes a given vertex must delete the vertex and all the edges incident to it. The number of edges can vary, but a vertex of degree 7 can be deleted in the same number of steps regardless of whether there are a total of 8 vertices or 100 vertices. The number of steps is constant when viewed as a function of the number of vertices. In this case, the algorithm *runs in constant time* or is a *constant-time algorithm*. Constant-time algorithms are the fastest, but often cannot perform very complicated procedures.

Suppose an algorithm A performs a maximum of $f(n)$ steps, where n is the number of vertices in the input graph. If $f(n)$ is a polynomial, then A *runs in polynomial time* or is a *polynomial-time algorithm*. Similar definitions exist for when f is linear, quadratic, cubic, and exponential. Linear-, quadratic-, and cubic-time algorithms are all examples of polynomial-time algorithms. Note that these definitions do not take into account any coefficients. Complexity is often used to determine how an algorithm will perform on a graph with a very large number of vertices. An algorithm that performs $f(n) = 10^6 n$ steps is prized over an algorithm that performs $g(n) = n^2$ steps because the former will be faster on all graphs with more than 10^6 vertices. Similarly, an exponential-time algorithm (no matter how small the base) will always end up running slower than a polynomial-time algorithm (no matter how large the degree), given a large enough graph.

In the context of determining whether a graph is embeddable on a surface, polynomial-time algorithms are considered "fast" (even when the degree of the polynomial is high) and

exponential-time algorithms are considered "slow." Linear-time algorithms are especially prized.

By implying the existence of Kuratowski-like and Wagner-like theorems for every surface S, the Robertson-Seymour Theorem also implies the existence of a polynomial-time algorithm that determines whether a given graph is embeddable on S. This algorithm simply looks for a subgraph of the given graph isomorphic to each topological obstruction (assuming they are all known), a polynomial-time process [24]. Since the number of topological obstructions is a finite constant k, and the algorithm that looks for a subgraph isomorphic to the obstruction runs in cubic time, this algorithm performs a maximum of $f(n) = kn^3$ steps. Besides the fact that determining the whole set of topological obstructions for a surface has proved difficult for most surfaces, it is also clear that the constant k is very large for all but the simplest surfaces. Therefore, the search has been for embeddability algorithms that use a different technique to achieve better results.

Planarity Checks and Toroidality Checks

Algorithms that determine whether a given graph is planar are called planarity checks. It is expected that these algorithms provide a plane embedding of a planar graph and exhibit a Kuratowski subdivision (or minor) in a non-planar graph. Linear-time planarity checks have been known since the 1970's. A more detailed history of planarity checks is given in the next chapter.

Toroidality checks are algorithms that determine whether a graph has an embedding on the torus. Again, these algorithms are expected to return a torus embedding of a toroidal graph. In 2002, Gagarin and Kocay [9] published a paper containing a linear-time toroidality check for $K_{3,3}$-free graphs. However, the algorithm presented is very theoretical and does not provide any means to extend to all graphs. In 2006, Woodcock's master's thesis [29] improves an existing exponential-time toroidality check. While the improvements make the algorithm practical (implementable), simpler, and faster, the result is still exponential-time.

The goal of this project is to find a practical (implementable), polynomial-time toroidal-

ity check. We have extended the simple and intuitive linear-time PC-Tree planarity check of Shih and Hsu [26] into a linear-time toroidality check for graphs that do not contain a subdivision of $K_{3,3}$. In the future, we hope to be able to extend this algorithm further to serve as a toroidality check for all graphs.

In Chapter 2 of this dissertation, there is a discussion of the PC-Tree algorithm of Shih and Hsu. Chapter 3 investigates various important results about $K_{3,3}$-free graphs, and Chapter 4 presents our extension of the PC-Tree algorithm. Finally, in Chapter 5, determining whether a graph can be covered with two Kuratowski subdivisions (useful for identifying toroidal obstructions) is proven to be NP-complete.

CHAPTER 2

PC-TREE ALGORITHM

2.1 – History

Linear-time planarity checking algorithms have been known since 1974 [13]. Remarkably, no linear-time toroidality checking algorithm has yet been found. Since then there have been several more linear-time planarity algorithms developed. Among these are path addition algorithms, vertex addition algorithms, and edge addition algorithms.

In 1976, Booth and Lueker [2] developed a data structure, called a PQ-Tree, to test whether a given matrix has the consecutive ones property (not defined here). Their algorithm was later adapted to test planarity in linear time. The PQ-Trees themselves seem particularly suited to be useful on graphs, but the algorithm uses a large amount of cases to reduce the tree, and these are not intuitive.

In 1999, Shih and Hsu [26] further developed the idea of PQ-Trees by introducing a similar but more flexible structure called a PC-Tree. This innovation simplified and clarified algorithms based on PQ-Trees. The PC-Tree algorithm remains one of the most intuitive (although complicated) planarity check algorithms. It is also powerful because it is capable of compactly describing all different planar embeddings of a planar graph. For these reasons, we chose the PC-Tree algorithm as our basis in an attempt to extend a planarity check algorithm into an algorithm to determine whether a graph can be drawn on the torus without crossing edges.

In 2001, Hsu and McConnell [15] wrote a chapter for the Handbook of Data Structures and Applications detailing the use of PQ-Trees for the consecutive ones problem and for determining whether a given graph is an interval graph. This chapter clearly describes the PC-Tree planarity algorithm in detail.

In 2003 Wen-Lian Hsu wrote a paper [14] describing an implementation of the PC-Tree Algorithm including pseudo-code for various parts of the algorithm. The abstract of this paper includes a link to a LEDA implementation of the PC-Tree Algorithm.

Finally, also in 2003, Boyer, Fernandes, Noma, and DePina [3] wrote a paper of corrections to the PC-Tree algorithm. These corrections filled small, but annoying, holes in the technical details of the algorithm. They also implemented the algorithm and compared its practical running time to some other known linear planarity algorithms, including the Booth-Lueker and Boyer-Myrvold algorithms. The PC-Tree algorithm was found to be the slowest of those tested. However, the structural advantages to the PC-Tree algorithm make it a likely candidate for extension to a practical, polynomial-time toroidality check.

The rest of this chapter is a complete description of the PC-Tree algorithm, distilling the information from the various sources listed above into one coherent explanation of the algorithm. This uses the vocabulary and notation from Hsu and McConnell [15]. Each section of this chapter deals with a portion of the algorithm. Section 2.2 deals with the ordering of the vertices via Depth First Search and Postorder Numbering. Section 2.3 introduces P-nodes and C-nodes. In section 2.4, discusses how the recursive steps of the algorithm work, including full and empty nodes, finding the terminal path, and updating the PC-Tree. The process of how the algorithm identifies a Kuratowski subdivision in a non-planar graph is in Section 2.5 and how it exhibits an embedding of a planar graph is in Section 2.6. One single example is followed all the way through these sections. Section 2.7 contains further examples, Section 2.8 provides a proof of correctness for the algorithm, and Section 2.9 has a brief discussion of the implementation of the PC-Tree data structure.

The graph depicted in Figure 2.1 is the example graph that will be revisited throughout the text to illustrate the PC-Tree algorithm and related concepts. Call this graph S. The PC-Tree algorithm provides a planar embedding of a graph, or provides a certificate of non-planarity in the form of a subgraph that is a subdivision of a Kuratowski graph.

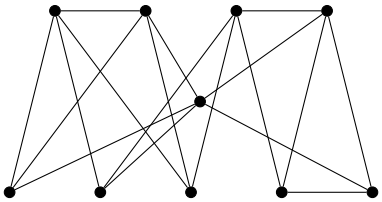

Figure 2.1: The example graph S.

2.2 – Depth First Search and Postorder Numbering

The PC-Tree Algorithm is a *vertex-addition algorithm*, meaning it adds a single vertex at each step of the algorithm. Therefore, the algorithm first needs to determine the order in which it will add the vertices. This is done using a *depth-first search* (DFS) tree. This process builds a spanning tree T of the graph G and also defines an order on the vertices of G. It starts at any vertex of G; this is called the *root* of the tree and it is the first vertex in the ordered list. At each vertex x, an arbitrary neighbor y of x that is not already in $V(T)$ is added to $V(T)$, entered as the next vertex in the list, and the edge from x to y is added to $E(T)$. If no such neighbor of x exists, the process considers the neighbors of z, the neighbor of x on the path in T from x to the root. It keeps moving up towards the root until it finds a neighbor that has not been added to $V(T)$. This continues until all vertices have been added. See Algorithm 1.

Algorithm 1 DepthFirstSearch

```
 1: procedure DepthFirstSearch(G)
 2:     VISITED ← ∅
 3:     v ← root
 4:     while VISITED ≠ V(G) do
 5:         if there is x ∈ N(v)\VISITED then
 6:             VISITED ← VISITED ∪{x}
 7:             v ← x
 8:         else if PARENT(v) exists then
 9:             v ← PARENT(v)
10:         else
11:             v is the root and the graph has been traversed
12:             return VISITED
```

The edges of G are thus partitioned into the edges of the depth-first search tree are called

tree edges and the remaining edges called *back edges*. A vertex with an incident back edge from one of its descendants is called a *back vertex*. The *back neighborhood* of a vertex v is all back edges from descendants of v to v.

The ordered list defined by the depth-first search of the graph G starts with the root and lists the vertices in the order they were visited. The PC-Tree Algorithm uses this list in reverse to define the order in which vertices of G are added to the PC-Tree. In other words, the first vertex to be added to the PC-Tree is the last vertex visited in the depth-first search and the root of the depth-first search tree is the last vertex added to the PC-Tree. This reverse depth-first search numbering is called a *postorder numbering*. Therefore the first vertex added to the PC-Tree is 1 and the last is n. Suppose the algorithm is at the step where it adds i. Then all the vertices from 1 to $i-1$ are *earlier* than i and all the vertices from $i+1$ to n are *later* than i.

In the example graph S, suppose the vertex second from the right in the bottom row is the root r of the DFS Tree. Figure 2.2 shows two possible DFS trees on S with root r. The vertices are labelled according to the postorder numbering. In both cases, the root is labelled 10 because there are 10 vertices in S. On the left, the DFS algorithm must backtrack after visiting vertices 7 and 3. On the right, the DFS Tree happens to be a Hamiltonian path. When future sections make reference to this example, the PC-Tree algorithm uses the DFS tree on the right, but the process would be similar if it used the tree on the left.

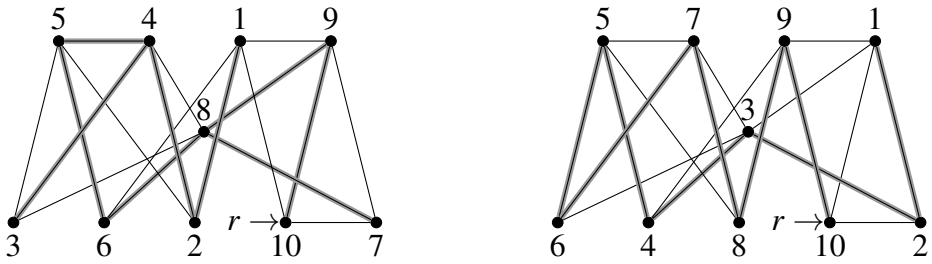

Figure 2.2: Two different DFS Trees on S. The DFS Tree on the right is used in future sections.

The reader should note that there are conflicting reports on whether the PC-Tree Al-

gorithm is a *vertex-addition* algorithm or an *edge-addition* algorithm. In [26], Shih and Hsu mention that they "add all back edges from the descendants to node *i*," indicating edge-addition. However, in Boyer and Myrvold's 2004 paper [4], they make reference to the "fundamental operation of adding a back edge" in their planarity algorithm. Realizing that the fundamental operation of the PC-Tree Algorithm is the addition of a *set* of all edges with a certain property in relation to one vertex, we treat the PC-Tree Algorithm as a vertex-addition algorithm.

2.3 – P-nodes and C-nodes

The PC-Tree algorithm creates an unrooted tree consisting of P-nodes and C-nodes called the PC-Tree. As the algorithm manipulates the tree, it follows certain rules. In order to find a rotation system of a planar embedding of the graph, the neighbors of a P-node (which correspond to the vertices of the graph) can be permuted in any way. However, the neighbors of a C-node (which correspond to certain 2-connected components of the graph) have a cyclic order that can only be reversed. P-nodes are represented with a small closed vertex and C-nodes are represented with a circled vertex. See Figure 2.3. The P-nodes in the PC-Tree are the vertices of the graph G, and the C-nodes represent the cycles and the boundaries, up to rotation, of planar embeddings of certain subgraphs of G.

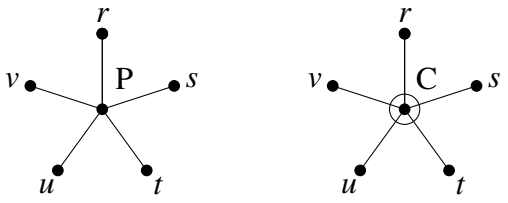

Figure 2.3: The neighbors of a P-node can be permuted in any way. The order of the neighbors of a C-node can only be reversed.

The PC-Tree algorithm is recursive; that is, it calls itself. The algorithm adds the back neighborhood of vertex *i*, reduces the PC-Tree, and then makes a call to itself to add the back neighborhood of the next later back vertex. It then receives either a Kuratowski subdivision

or an embedding of all back neighborhoods of vertices later than i. If it receives a Kuratowski subdivision, it simply returns that. If it receives a partial embedding, it extends that embedding by including the back neighborhood of i and returns the extended embedding.

2.4 – Full and Empty Nodes and the Terminal Path

The PC-Tree is initialized as the DFS tree T of G. Recall that a vertex i is *earlier* than vertex j if $i < j$ in the postorder numbering of $V(G)$. In this case, j is *later* than i. Recall that the *back neighborhood* of a vertex v is all back edges from descendants of v to v. Before adding the back neighborhood of vertex i, all earlier neighbors of i in G are marked as *full* in the PC-Tree and all earlier vertices that have a neighbor in G later than i are marked as *empty*. Vertices that are both full and empty get marked as *partial*.

Any remaining internal vertices in the PC-Tree (including any C-nodes) that are earlier than i are classified as follows. If there exists a rooting of the subtree induced by the descendants of i in which the subtree of vertex x consists entirely of full (empty) nodes, then x is full (empty). If no such rootings exist for a vertex, or if a vertex would be classified as both full and empty, then that vertex is classified as partial.

An edge of the PC-Tree is a *terminal edge* if both of its endpoints are partial. Equivalently, an edge is terminal if removing it results in two subtrees, each with both full and empty nodes. It is clear that the terminal edges must be connected. If the terminal edges form a path, then this is called the *terminal path*.

If, for each vertex on the terminal path, there is a legal ordering of its neighbors of the form

$$\text{(terminal), full, } \ldots \text{, full, (terminal), empty, } \ldots \text{, empty,}$$

then planarity is maintained and the algorithm uses the terminal path to create a C-node. This is proved in Lemma 2.1. Note that either (or both) terminal neighbors might need to be ignored if the vertex is an endpoint (or the only vertex) of the terminal path. Note that since P-nodes accept any permutation of their neighbors, only C-nodes (specifically, those

of degree at least 4) on the terminal path can create a non-planarity if the terminal edges form a path.

Suppose the algorithm is at the step where it adds the back neighborhood of vertex i and a cycle is created by doing this. Suppose further that terminal edges form a path and that the full and empty subtrees can be flipped onto opposite sides of the terminal path. In order to create a C-node, the PC-Tree algorithm needs to identify which vertices become neighbors of the C-node, and which vertices and edges are removed from the PC-Tree. As a rule, all full nodes not on the terminal path are removed. This includes any nodes on the paths from the endpoints of the terminal path to the vertex i and all the nodes in the full subtrees. Any vertices on the terminal path that are not empty and do not have any empty neighbors are *smoothed*, that is, such a vertex is deleted and its neighbors on the terminal path become adjacent. Once these deletions are complete, the terminal edges and any edges incident to i are removed and the C-node created. The C-node is adjacent to the remaining vertices of the terminal path (in order), and vertex i. If the new C-node ends up adjacent to another C-node from the terminal path, the edge between them is contracted and the neighborhood of the older C-node becomes part of the neighborhood of the new one, preserving the order and position of the neighbors.

Presented below are most of the steps of the PC-Tree algorithm working on the example graph S and most of the reductions explained in the previous paragraph are put into action. Recall that the algorithm is using the DFS postorder numbering given on the right side of Figure 2.2. The PC-Tree is initialized as the DFS Tree, which happens to be a path.

Vertex 3 is the earliest vertex with a back neighborhood (edges to descendants). Both vertices 1 and 2 are neighbors of 3, therefore they are full (denoted with a black triangle). However, they also both have neighbors later than 3, therefore they are also empty (denoted with a white triangle). Since vertices 1 and 2 are both full and empty, they are designated as partial, and the edge between them constitutes the terminal path at this step (highlighted in gray). Since both vertices on the terminal path are empty, they become neighbors of

the soon-to-be-created C-node. Edges 12, 23, and 13 are deleted, and the new C-node is adjacent to vertices 1, 2, and 3. This is the simplest way in which a C-node is created. Note that there is a second white triangle at 1 only as a reminder that it has two neighbors later than 3 (namely, 9 and 10). Please see Figure 2.4 for a visualization of this paragraph.

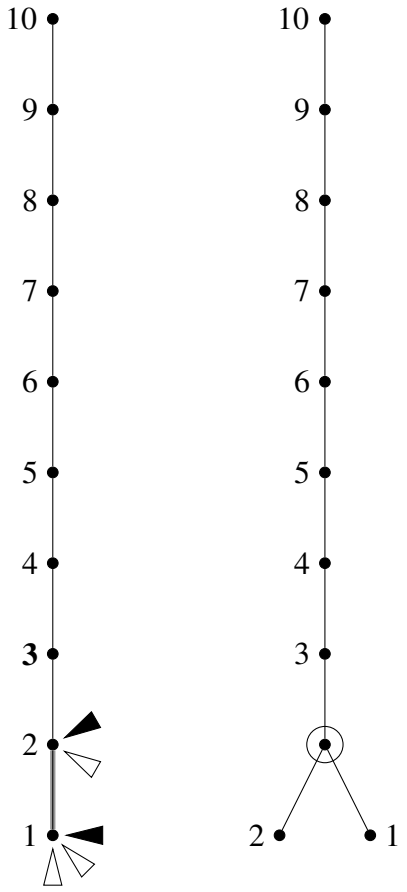

Figure 2.4: The PC-Tree for S before and after adding the back neighborhood of vertex 3.

Vertices 4 and 5 do not have back neighborhoods. Therefore, no edges are added and no C-node created. Vertex 6 is the next vertex with a back neighborhood. The neighbors of 6 are 3 and 5, therefore, they are full. All of the vertices in $\{1, 2, 3, 4, 5\}$ are empty since they all have a neighbor later than 6. So, vertices 3 and 5 are partial. In order to complete the classification of vertices, consider the subtree induced by the descendants of 6, that is the vertices 1, 2, 3, 4, 5, and the C-node. If this tree is rooted at 3, then the subtree at 3 consists of 1 and 2, both empty. Therefore, 3 is empty. There is no rooting of this tree so that the

subtree at 4 is all full or all empty. Therefore, 4 is partial. Equivalently, deleting 3 leaves a component of all empty nodes (so 3 is empty), but deleting 4 leaves two components, both with full and empty nodes (so 4 is partial). The edges 34 and 45 form the terminal path, and all vertices on the terminal path are empty, so none are deleted or smoothed. These edges are removed along with edge 56 and another C-node is created, adjacent to 3, 4, 5, and 6 (in that clockwise order). Note that the subtree at 3 does not change. Please see Figure 2.5 for a visualization of this paragraph.

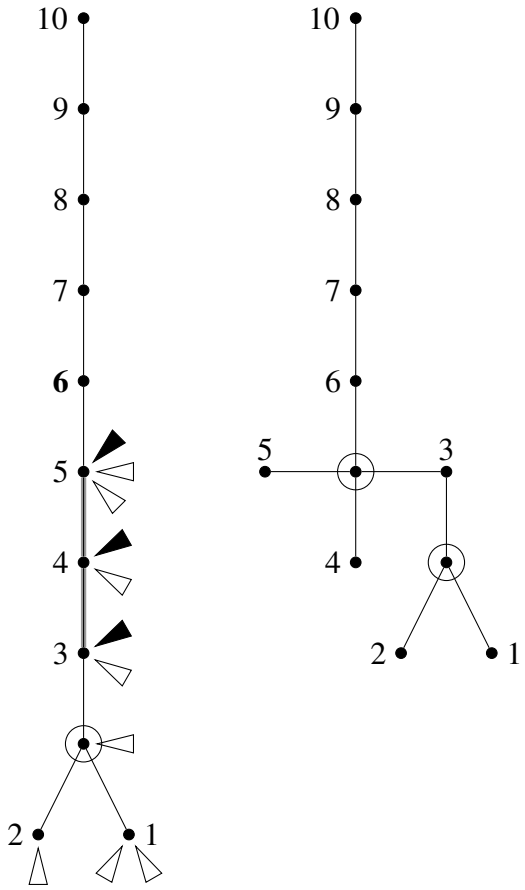

Figure 2.5: The PC-Tree for S before and after adding the back neighborhood of vertex 6.

Vertex 7 has the next back neighborhood. Vertices 3, 5 and 6 are neighbors of 7, and so they are full. Vertices 1, 2, 4, and 5 all have neighbors later than 7, so they are all empty. Vertex 3 is empty since it has an empty subtree. Therefore 3 and 5 are partial. The C-node adjacent to 1, 2, and 3 is empty for the same reason it was in the last step. The other C-node

is partial because deleting it from the subtree induced by the descendants of 7 leaves an empty subtree (4) and a full subtree (6). Therefore, the edges from the C-node of degree four to 3 and to 5 form the terminal path. Note that vertex 6 is deleted because it is a full node not on the terminal path. It is clear that the full subtrees can be flipped to one side of the terminal path and the empty can be flipped to the other. A third C-node is created adjacent to 3, the C-node that had been on the terminal path, 5, and 7 (in that clockwise order). The edge between the C-nodes is contracted, resulting in a C-node adjacent to 3, 4, 5, and 7 (again, in that clockwise order). Please see Figure 2.6 for a visualization of this paragraph.

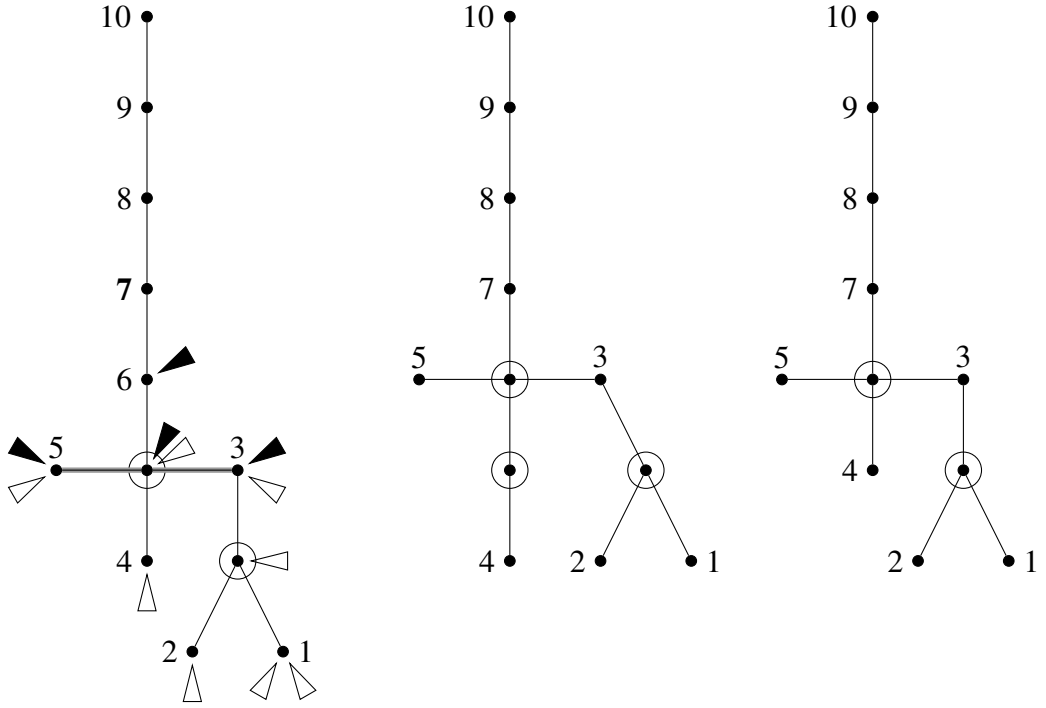

Figure 2.6: The PC-Tree for S before and after adding the back neighborhood of vertex 7, then after combining the adjacent C-nodes.

Vertex 8 has the next back neighborhood. Vertices 5 and 7 are adjacent to 8, and so they are full. Vertices 1, 2, and 4 are all empty since they have neighbors later than 8. The C-node adjacent to 1, 2, and 3 and vertex 3 are empty for the same reasons they have been in past steps. The other C-node is partial because deleting it from the subtree induced by

the descendants of 8 leaves both entirely full and entirely empty trees. Here is a case where there is only one partial node. This is considered a terminal path of length 0. Since the full and empty subtrees are not alternating around the C-node, the full can be flipped to one side of the "terminal path" and the empty to the other side. The vertices 5 and 7 are deleted since they are full vertices not on the terminal path. Therefore a C-node adjacent to 8 and the terminal C-node is created. The C-nodes are immediately combined into a C-node adjacent to 3, 4, and 8 (in that clockwise order). Please see Figure 2.7 for a visualization of this paragraph.

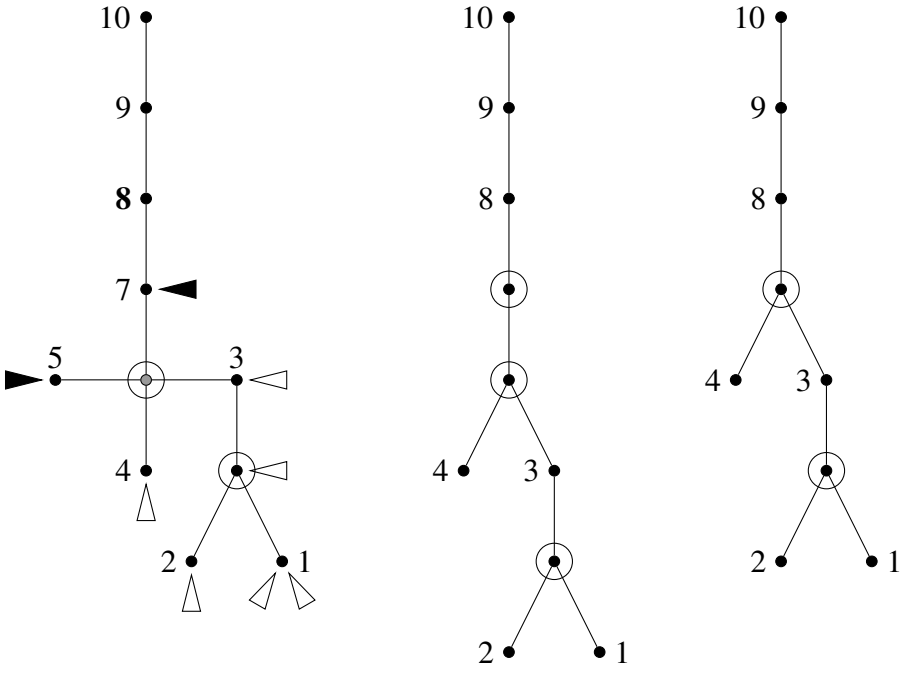

Figure 2.7: The PC-Tree for S before and after adding the back neighborhood of vertex 8, then after combining the adjacent C-nodes.

Vertex 9 has the next back neighborhood. Vertices 1, 3, 4, and 8 are full since they are adjacent to 9. Deleting the C-node adjacent to 8 from the subtree induced by the descendants of 9 leaves a full subtree (8), so it is full. Vertices 1 and 2 are empty since they have neighbors later than 9. Therefore, 1 is partial and since deleting the C-node adjacent to 1 from the subtree induced by the descendants of 9 leaves a full tree and an empty tree, the C-node is also partial. So, the edge incident to 1 is the terminal path and clearly the full and partial

subtrees can be flipped to opposite sides. Vertices 3, 4, and 8, as well as the full C-node are all deleted since they are full and not on the terminal path. A new C-node is created and is adjacent to 1, the partial C-node, and 9 (in that clockwise order). The two C-nodes are combined to create a C-node adjacent to 1, 2, and 9 (in that clockwise order). Please see Figure 2.8 for a visualization of this paragraph.

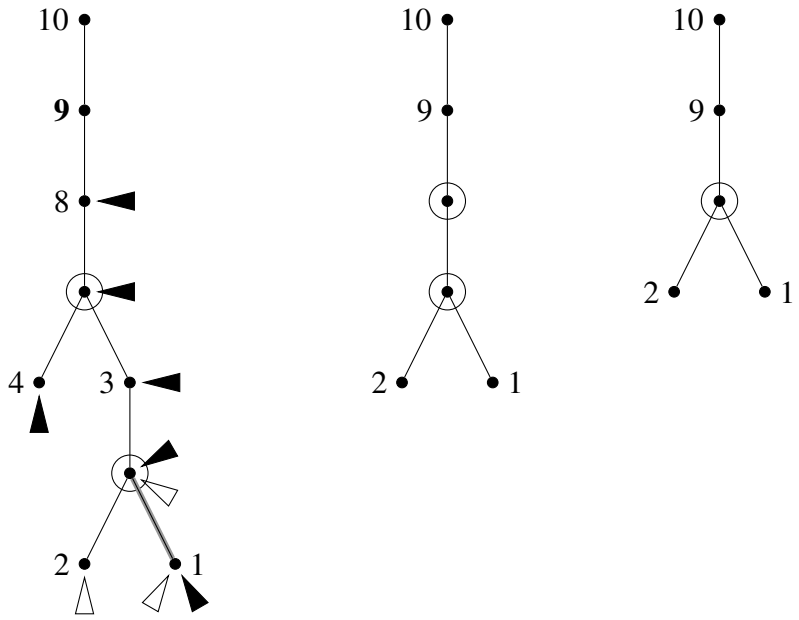

Figure 2.8: The PC-Tree for S before and after adding the back neighborhood of vertex 9, then after combining the adjacent C-nodes.

The final back neighborhood is the back neighborhood of vertex 10. All other vertices of the PC-Tree are full since there are no vertices later than 10. As discussed in the next section, a Kuratowski subdivision cannot be detected when adding the last vertex of the graph. Therefore, the PC-Tree algorithm concludes that the graph is planar since it did not find any Kuratowski subdivisions. A later section discusses how the algorithm extracts an embedding of a planar graph given the PC-Tree, including one for the example graph S. However, the configurations in the PC-Tree that correspond to Kuratowski subdivisions in non-planar graphs are the next topic.

2.5 – Finding a Kuratowski Subdivision

This section consists of the proof of one critical result, stated below. However, the proof also provides the exact configurations for which the PC-Tree algorithm searches in order to find a subdivision of $K_{3,3}$ or K_5 in a non-planar graph.

Lemma 2.1. If the input graph G is planar, the terminal edges at each step of the PC-Tree algorithm form a path (or a single vertex) such that for each vertex on the terminal path, there is a legal ordering of its neighbors where the full and empty subtrees are on opposite sides of the terminal path.

Proof. Argue by proving the contrapositive: If the terminal edges do not form a path, or if the full and empty subtrees cannot be flipped to opposite sides of the terminal path, then the graph is not planar. In particular, the PC-Tree algorithm can then identify a subdivision of $K_{3,3}$ or K_5.

Suppose first that the terminal edges do form a path (or there are no terminal edges), but that there is a terminal C-node with neighbors (a, b, c, d) in that order where a and c are full and b and d are empty. Then there are edges from b and d to vertices later than i. Let t be the earliest of these vertices. Then $\{a, c, i\}$ and $\{b, d, t\}$ are the partite sets of branch vertices of a $K_{3,3}$ subdivision. See Figure 2.9.

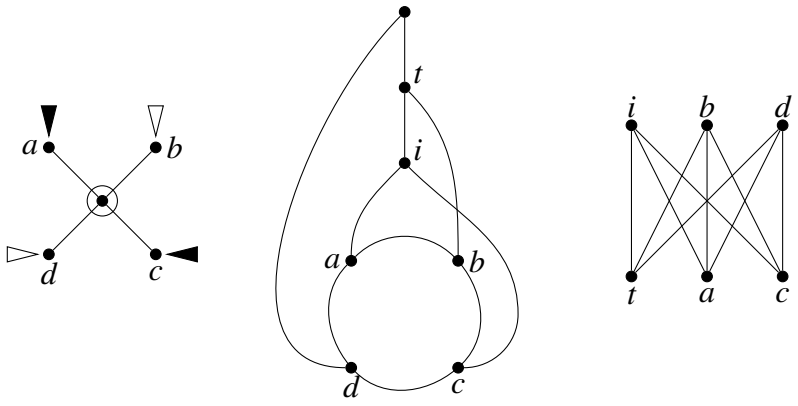

Figure 2.9: A subdivision of $K_{3,3}$ from a C-node with four alternating full and empty neighbors.

Now suppose that the terminal edges do not form a path. Then they form a subtree with

at least three leaves, z_1, z_2, and z_3, each of which has at least one full subtree and at least one empty subtree. Let w be the vertex so that the paths $z_1 - w$, $z_2 - w$, and $z_3 - w$ are edge-disjoint. Since z_k has an empty subtree, there is at least one path from z_k to a vertex later than i. For each $k \in \{1, 2, 3\}$, let t_k be the earliest of these vertices. Note that t_1, t_2, and t_3 need not be distinct. Let t be the vertex whose DFS label is the median of the DFS labels of t_1, t_2, and t_3.

If w is a P-node, then $\{z_1, z_2, z_3\}$ and $\{t, i, w\}$ are the partite sets of branch vertices of a $K_{3,3}$ subdivision. See Figure 2.10.

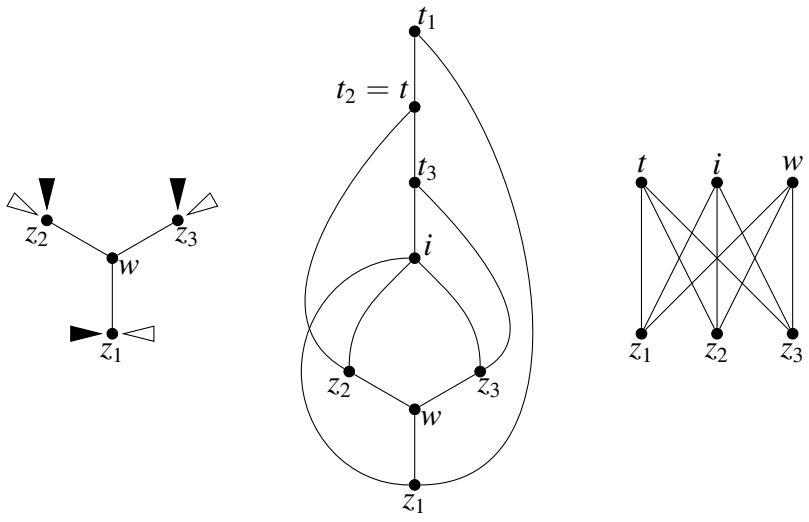

Figure 2.10: A subdivision of $K_{3,3}$ from three terminal edges incident to a P-node.

Suppose w is a C-node. If at least one of z_1, z_2, and z_3 is not a neighbor of w (without loss of generality, say z_3), then there is a neighbor of w, called w' with paths to z_1 and z_2 in G (since w', z_1, and z_2 are neighbors of a C-node) and a path to z_3 in the PC-Tree (and hence G). Then $\{z_1, z_2, z_3\}$ and $\{t, i, w'\}$ are the partite sets of branch vertices of a $K_{3,3}$ subdivision. See Figure 2.11.

Now consider the case where w is a C-node, and all of z_1, z_2, and z_3 are neighbors of w. Without loss of generality, t_3 is a minimal element of t_1, t_2, and t_3. If either t_1 or t_2 is equal to t_3, then $\{z_1, z_2, z_3, i, t_3\}$ is the branch vertex set of a K_5 subdivision. See Figure 2.12.

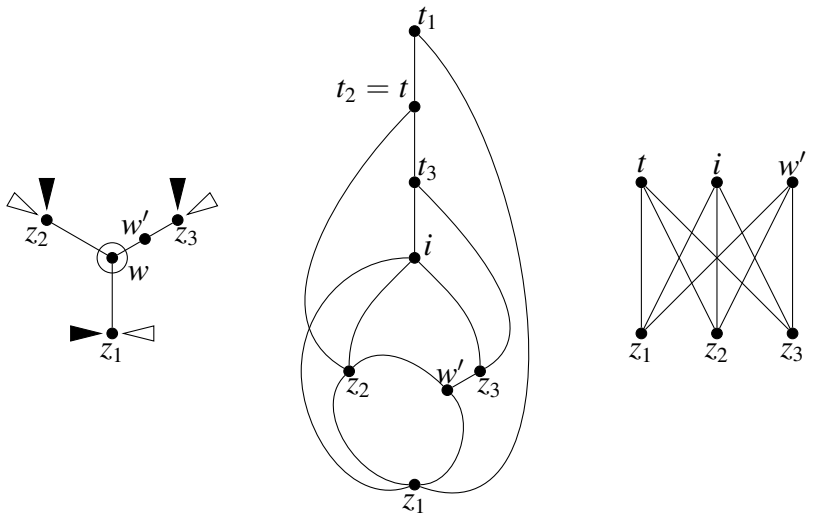

Figure 2.11: A subdivision of $K_{3,3}$ from three terminal edges incident to a C-node where one of the neighbors of the C-node is incident to another terminal edge.

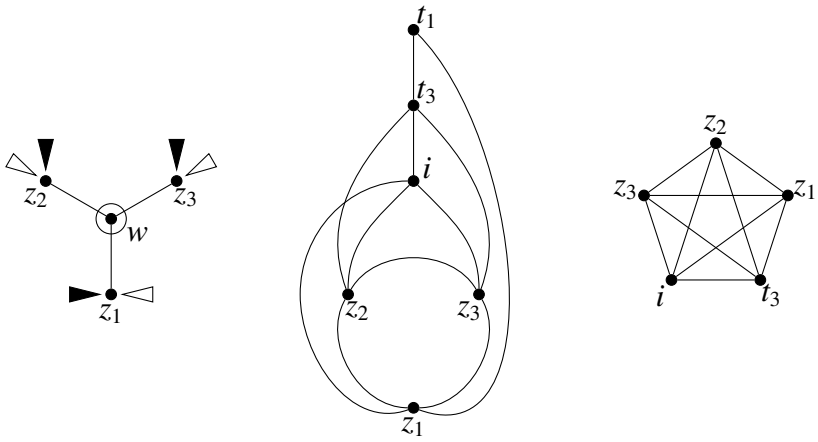

Figure 2.12: A subdivision of K_5 from three terminal edges incident to a C-node where the earliest back ancestor of i is adjacent to at least two of the subtrees of the neighbors of the C-node.

If $t_1 \neq t_3 \neq t_2$, then let $t = \min\{t_1, t_2\}$. Then $\{i, t, z_2\}$ and $\{z_1, t_3, z_3\}$ are the partite sets of branch vertices of a $K_{3,3}$ subdivision. See Figure 2.13.

Since Kuratowski subdivisions arise in these situations, G is non-planar, and the proof is complete. □

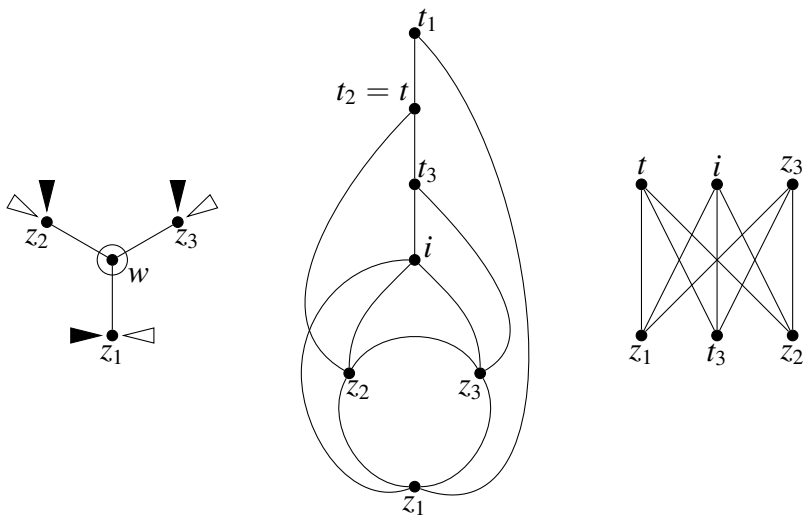

Figure 2.13: A subdivision of $K_{3,3}$ from three terminal edges incident to a C-node where the earliest back ancestor of i is adjacent to only one of the subtrees of the neighbors of the C-node.

2.6 – Producing Embeddings

If the back neighborhoods of all the vertices of G get added without encountering a Kuratowski configuration, then G must be planar. However, planarity algorithms should be demonstrative, that is, they should exhibit a planar embedding when the graph is planar. One of the advantages of the PC-Tree algorithm is that it is actually able to produce all possible planar embeddings of a planar graph.

If the back neighborhood of $n-1$ has been added and the PC-Tree reduced without encountering any Kuratowski configurations, then the algorithm concludes that the graph G is planar. The back neighborhood of n can be trivially embedded (possibly in many ways). This is the first partial embedding that gets returned to the calling function. Thus the embedding process takes place as the algorithm steps back out of the recursive calls.

After adding the back neighborhood of vertex i and reducing the PC-Tree, the recursive function calls itself to add the neighborhood of vertex $i+1$. Suppose this call returns an embedding of all later back neighborhoods. Then the function can extend this embedding by embedding the back neighborhood of i since no Kuratowski subdivision was returned. This extended embedding gets returned to the calling function.

This process is probably best illustrated by following it through our example graph S. At the end of Section 2.4, the PC-Tree algorithm paused its worked on the graph S having added the back neighborhood of vertex 9, and finding no Kuratowski configuration, declared S planar. From this point, it produces an embedding of S in the plane. Recall that the PC-Tree algorithm can produce any of the planar embeddings of a graph. Presented below are two tracks for the PC-Tree algorithm to take, each producing a different embedding of S. This split is based on a decision in the first embedding step. At several points after that, the algorithm is shown choosing just one embedding of the "active" back neighborhood, when a different choice could be made to produce a different embedding. These options arise because the order of incident edges at P-nodes can be permuted in any way and the order at C-nodes can be reversed.

Embedding the back neighborhood of vertex n is the base case for the recursive function. For S, this means embedding the edges (10)1 and (10)2. Because the order of the edges incident to the C-node can be reversed, the algorithm can embed the edges in two different ways, shown in Figure 2.14.

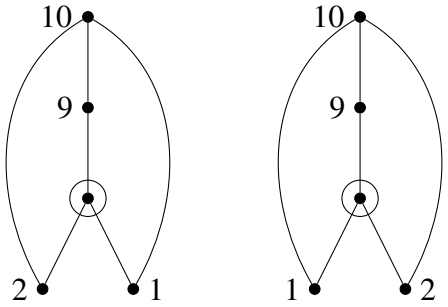

Figure 2.14: Embedding the back neighborhood of 10 in two different ways.

The choice between these two options determines which partial embedding gets returned, and therefore, which partial embedding gets extended to the embedding of S. The consequences of both choices are continued here, keeping the choices consistent whenever possible in the future to exemplify how two different embeddings can be obtained from the same PC-Tree.

Next the algorithm embeds the back neighborhood of 9. The choice of embedding in the previous step forces the algorithm to embed the edges incident to 9 in a different order. See Figure 2.15.

Now the algorithm embeds the back neighborhood of 8. Notice that it is using the version of the PC-Tree that was current just after adding the back neighborhood of 8 and reducing the tree. See Figure 2.16.

Then the algorithm embeds the back neighborhood of 7. See Figure 2.17.

Next the algorithm embeds the back neighborhood of 6. Note that the edges 34 and 45 are present in the PC-Tree at this step. They are not drawn them vertically simply to make the picture clearer. See Figure 2.18.

Finally, the algorithm embeds the back neighborhood of 3. The results are in fact two

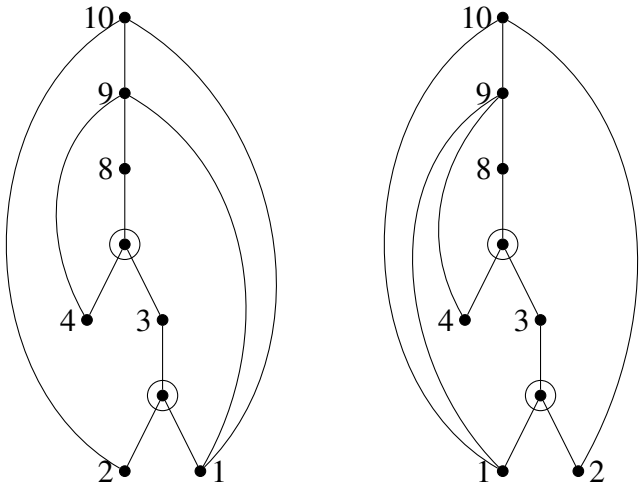

Figure 2.15: Embedding the back neighborhood of 9.

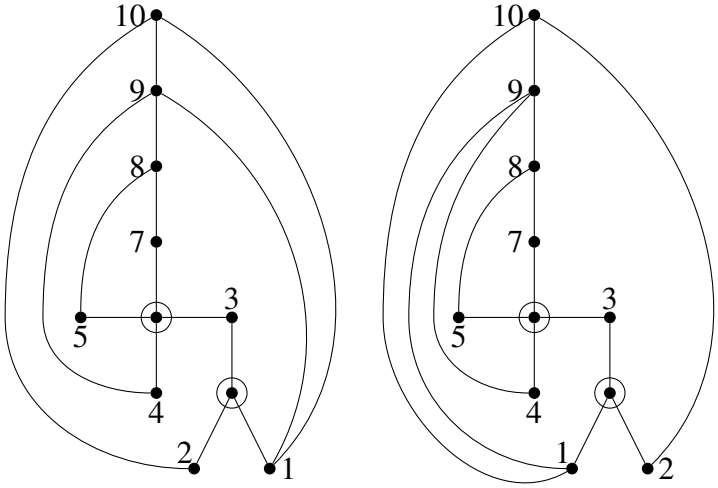

Figure 2.16: Embedding the back neighborhood of 8.

different embeddings of S. This can be verified by comparing the sizes of faces. For example, the face that is bordered by the edges 78 and 89 has size 5 on the left and size 6 on the right. See Figure 2.19.

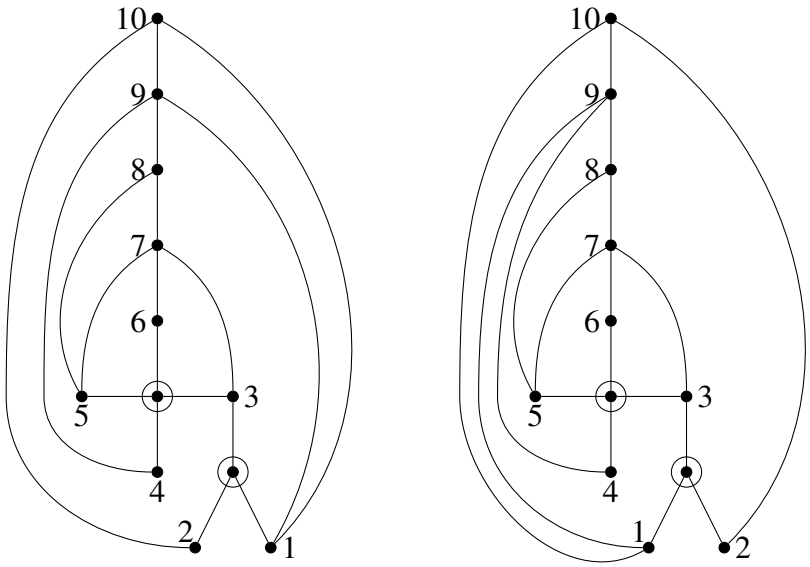

Figure 2.17: Embedding the back neighborhood of 7.

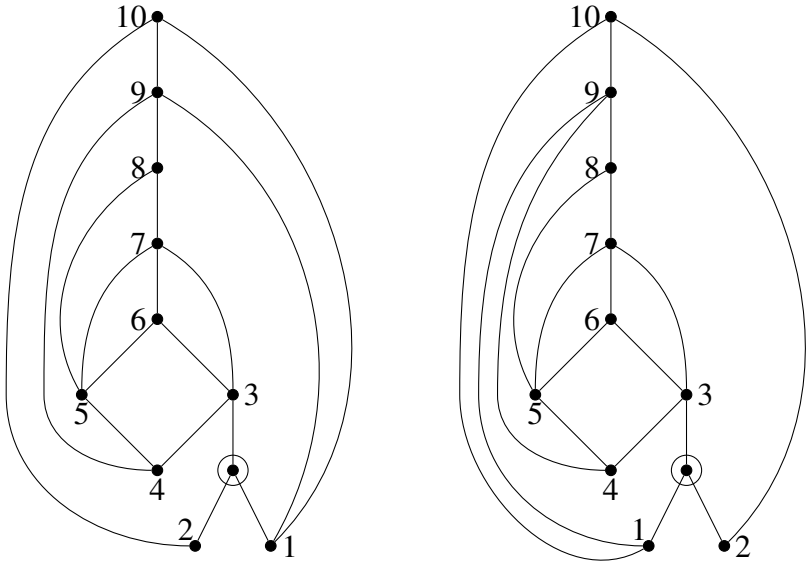

Figure 2.18: Embedding the back neighborhood of 6.

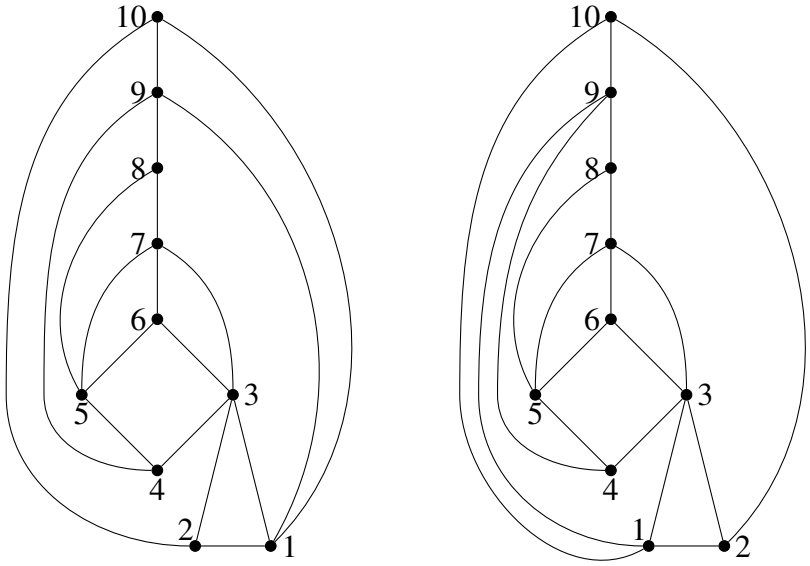

Figure 2.19: Embedding the neighborhood of vertex 3. The results are two different embeddings of S.

2.7 – Further Examples

Example 1–Planar

The first example is a graph that is obviously planar. The graph in Figure 2.20 is a subdivision of $K_{3,3}$ minus an edge. There are no vertices of degree four, so there is clearly a subdivision of neither K_5 nor $K_{3,3}$ in the graph.

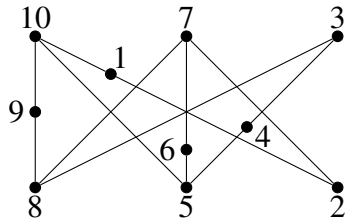

Figure 2.20: Example 1. A planar graph.

The vertices are numbered according to a post-order depth-first-search tree. As the algorithm begins to construct the PC-Tree, there are no cycles created by vertices 1 through 7, resulting in the diagram on the left in Figure 2.21. When the algorithm adds vertex 8, it finds that vertices 3 and 7 are full and vertices 1 and 5 are empty. Thus 4 is also full and 2 is empty. Vertices 5, 6, and 7 are partial since they are both full and empty. Thus there are two edges in the terminal path. The diagram on the right in Figure 2.21 shows the terminal path drawn horizontally with all the full subtrees flipped above and all the empty subtrees flipped below.

Now a C-node is created. All full nodes that are not on the terminal path are deleted (or embedded). Vertex 6 is also deleted since it is not empty not adjacent to an empty vertex. The new C-node is adjacent to the new vertex, 8, and to the remaining vertices of the terminal path, 5 and 7. Note that the cycle (2-connected component) consisting of the terminal path, the full subtrees, and the edges going to the new vertex 8 is removed.

No cycle is created by adding 9 to the PC-Tree, and now the algorithm has reached the final vertex of the graph to be added, 10. See the left diagram in Figure 2.22. All vertices of

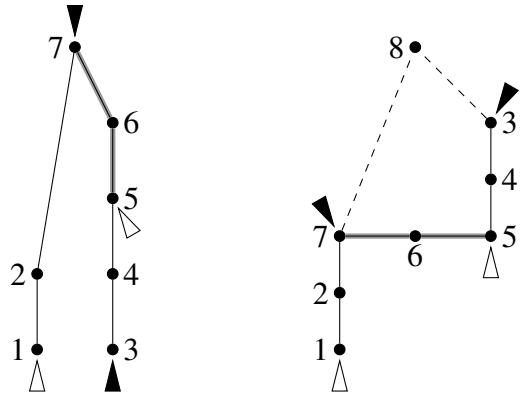

Figure 2.21: Example 1. The PC-Tree when adding vertex 8.

the PC-Tree are labelled as full and the edges can be added to the embedding trivially. See the center diagram in Figure 2.22. To obtain an embedding of the original graph, the PC-Tree algorithm need only expand the C-node back into the cycle (2-connected component) that it represents. See the right diagram in Figure 2.22. This is a planar embedding of the original graph.

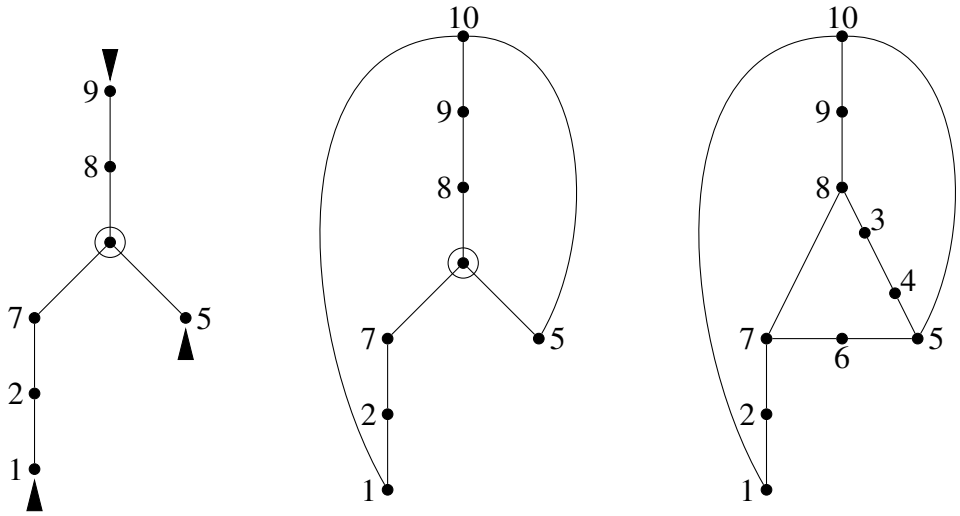

Figure 2.22: Example 1. The PC-Tree when adding vertex 10. A planar embedding of the original graph.

Example 2–Non-planar, $K_{3,3}$ one way

For the next example, add the edge 23 to the previous example to create a graph that has a $K_{3,3}$-subdivision. The numbering used in Example 1 is still a DFS numbering. See Figure 2.23.

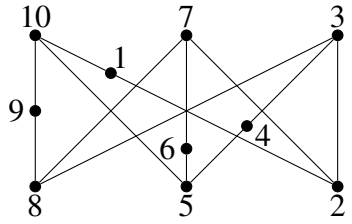

Figure 2.23: Example 2. A graph with a $K_{3,3}$-subdivision.

As the algorithm adds vertices to the PC-Tree, it creates no cycles until vertex 7. See the left side of Figure 2.24. The edges 23, 34, and 45 are the terminal edges because removing any one of them leaves components with both full and empty vertices. Orienting the terminal path horizontally, flipping the full vertices above the path, and the empty vertices below, it is clear how the cycle containing the terminal path and the vertex 7 can be embedded. See the center of Figure 2.24. Note that the full vertex 6 is removed from the tree because it can be trivially embedded and does not interact with any later vertices.

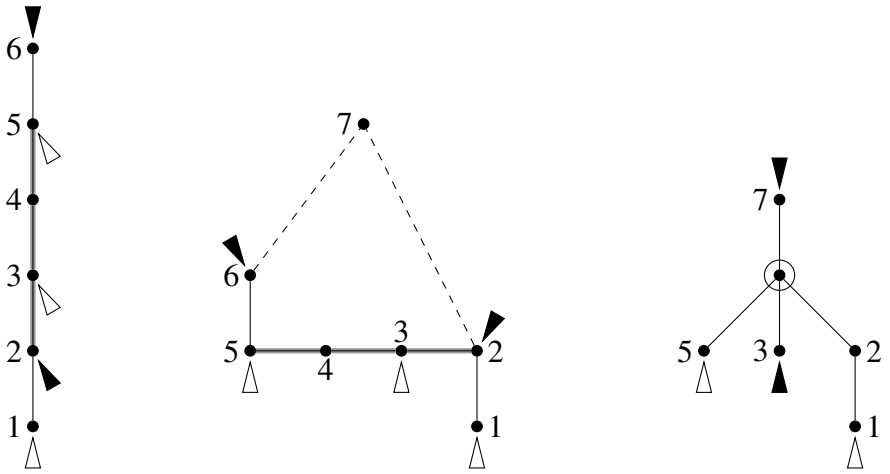

Figure 2.24: Example 2. The PC-Tree when adding vertex 7, the creation of the C-node, and the PC-Tree when detecting the $K_{3,3}$-subdivision by adding vertex 8.

The result is the PC-Tree on the right side of Figure 2.24. When adding vertex 8, the algorithm finds that there are no terminal edges, but just a single terminal vertex, the C-node. The full and empty vertices alternate in the cyclic order around the C-node. Therefore, this is the situation described in Figure 2.9.

There is a $K_{3,3}$-subdivision with partite sets $\{7, 3, 8\}$ and $\{5, 2, 10\}$. Note that $i = 8$ and $t = 10$ when referring to Figure 2.9. This subdivision was detected when attempting to add the fifth branch vertex.

Example 3–Non-planar, $K_{3,3}$ another way

In this example, add two more edges, 37 and 25, to the previous example. As before, the numbering is still a DFS numbering. This graph also contains a $K_{3,3}$-subdivision, but the PC-Tree algorithm finds a different configuration this time.

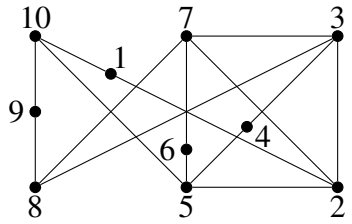

Figure 2.25: Example 3. Another graph with a $K_{3,3}$-subdivision.

No cycles are created until the PC-Tree algorithm adds vertex 5. The left side of Figure 2.26 shows the PC-Tree and the single terminal edge 23. Orienting the terminal edge horizontally with the full vertices above and the empty ones below, the embedding of the cycle that contains the edge 23 and vertex 5 becomes clear. See the center of Figure 2.26. As with vertex 6 in Example 2, vertex 4 gets removed from the tree.

The result is the PC-Tree on the right of Figure 2.26. When adding vertex 7, there is only one terminal vertex, the C-node. The algorithm has encountered the configuration in Figure 2.13. There is a $K_{3,3}$-subdivision with partite sets $\{7, 10, 3\}$ and $\{2, 5, 8\}$. Using the notation from Figure 2.13, $i = 7$, $t_3 = 8$, and $t = 10$. Also, z_1, z_2, z_3 are all adjacent to the C-node w, and $t_3 = 8$ is adjacent to only one neighbor of the C-node. This subdivision was

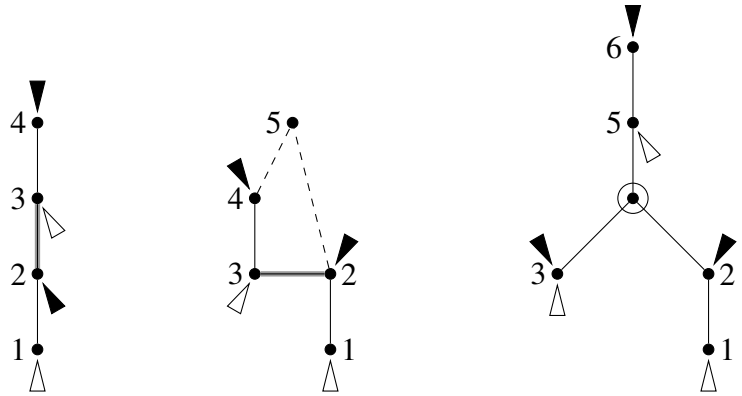

Figure 2.26: Example 3. The PC-Tree when adding vertex 5, creating the C-node, and the PC-Tree when adding vertex 7.

detected when attempting to add the fourth branch vertex.

Example 4–Non-planar, K_5

For the last example, add one more edge, 58, to the previous example. This clearly still has a $K_{3,3}$-subdivision, but it also now contains a K_5-subdivision. The numbering is still a DFS numbering and the algorithm detects the K_5-subdivision. See Figure 2.27.

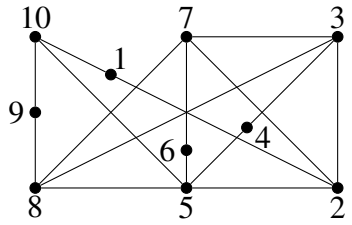

Figure 2.27: Example 4. A graph with both a $K_{3,3}$-subdivision and a K_5-subdivision.

No cycles are created until the algorithm adds vertex 5. The steps are the same as in Example 3, until adding vertex 7. See the left side and center of Figure 2.28. The PC-Tree looks exactly the same as in Example 3, but now vertex 5 is adjacent to 8.

Therefore, the PC-Tree algorithm encounters the configuration in Figure 2.12. There is a K_5-subdivision with branch vertex set $\{2, 3, 5, 7, 8\}$. Using the notation from Figure 2.12, $i = 7$, and $t = 8$. Also, z_1, z_2, z_3 are all adjacent to the C-node w, but $t_3 = 8$ is adjacent to two neighbors of the C-node. This subdivision was detected when attempting to add the

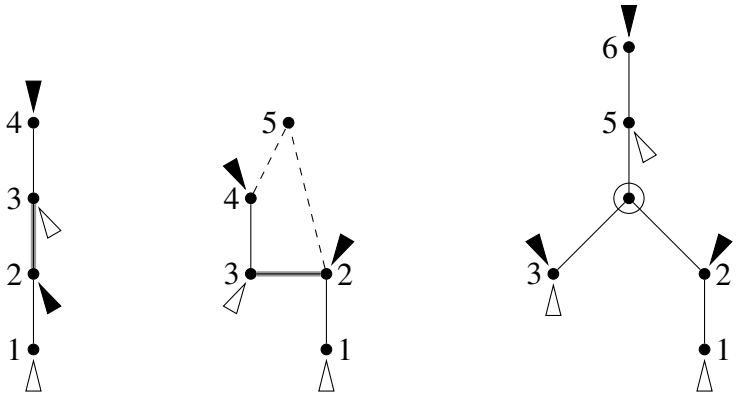

Figure 2.28: Example 4. The PC-Tree when adding vertex 5, creating the C-node, and the PC-Tree when adding vertex 7.

fourth branch vertex.

2.8 – Proof of Correctness

This section contains a proof of the following:

Theorem 2.2. The PC-Tree algorithm correctly identifies planar and non-planar graphs, producing an embedding of a planar graph, or a Kuratowski subdivision in a non-planar graph.

Proof. If G is planar, then the PC-Tree algorithm does not encounter any of the configurations in Section 2.5 by Lemma 2.1. Therefore, it adds all the vertices of G to the PC-Tree and then steps back out of the recursive calls, building a plane embedding of G.

If G is non-planar, then it contains a Kuratowski subdivision. Suppose that the PC-Tree algorithm does not find any of the configurations in Section 2.5. Then the algorithm adds all the vertices of the graph to the PC-Tree and begins to step back out of the recursive calls, building a plane embedding. However, at each step of the embedding process, the algorithm can expand any C-node into a trivially embeddable subgraph of the PC-Tree (due to all C-nodes being created only when the full and empty subtrees can be flipped to opposite sides of the terminal path). This cannot fail to produce a plane embedding of G. This contradicts the assumption that G is non-planar. Therefore, the PC-Tree algorithm must find one of the configurations in Section 2.5. □

2.9 – Implementation

Because the PC-Tree Algorithm adds each vertex one at a time and never backtracks, it is a linear-time algorithm. However, it is not fully clear that the analysis, updating, and maintenance of the PC-tree structure can be done in linear time. In order to show why these subprocesses run in linear time, a discussion of the implementation of the PC-Tree data structure is necessary.

The following paragraph is taken directly from [15] (Definition 1.1).

> The data structure for the PC tree is the following. Each P node carries a pointer to the parent edge. Each edge uv is implemented with two oppositely directed *twin arcs* (u, v) and (v, u). Each arc (x, y) has a pointer to its two neighbors in the cyclic order about y, a pointer to its twin, and a *parent bit* label that indicated whether y is the parent of x. In addition, if y is a P node, then (x, y) has a pointer to y. The algorithm chooses not to use an explicit representation of a C node; its existence is implicit in the doubly-linked circular list of its edges that gives their cyclic order. No two C nodes are adjacent, so each of these edges has one end that identifies a neighbor of the C node, and another end that indicates that the end is incident to a C node, without identifying the C node.

Figure 2.29 shows a diagram of the implementation of the indicated PC-Tree. Notice that there is no explicit C-node, just a doubly-linked list of arcs surrounding it. There is also a doubly-linked list of arcs around vertex 3, but since it is a P-node, there are pointers to and from it explicitly. The double-loop arcs such as those to and from $(3, 2)$ are there simply because one vertex has only one neighbor (such as 2). Therefore, the edge is adjacent to itself in both directions in the list of neighbors of that vertex.

These doubly-linked cyclic lists of edges are *discordant lists*, a type of list where there is an order to the elements, but no direction to that order. Each element has two pointers (to its two neighbors in the list). Whenever the algorithm traverses this list, it simply moves in the direction of the pointer that does not indicate the element it last visited. This change in the

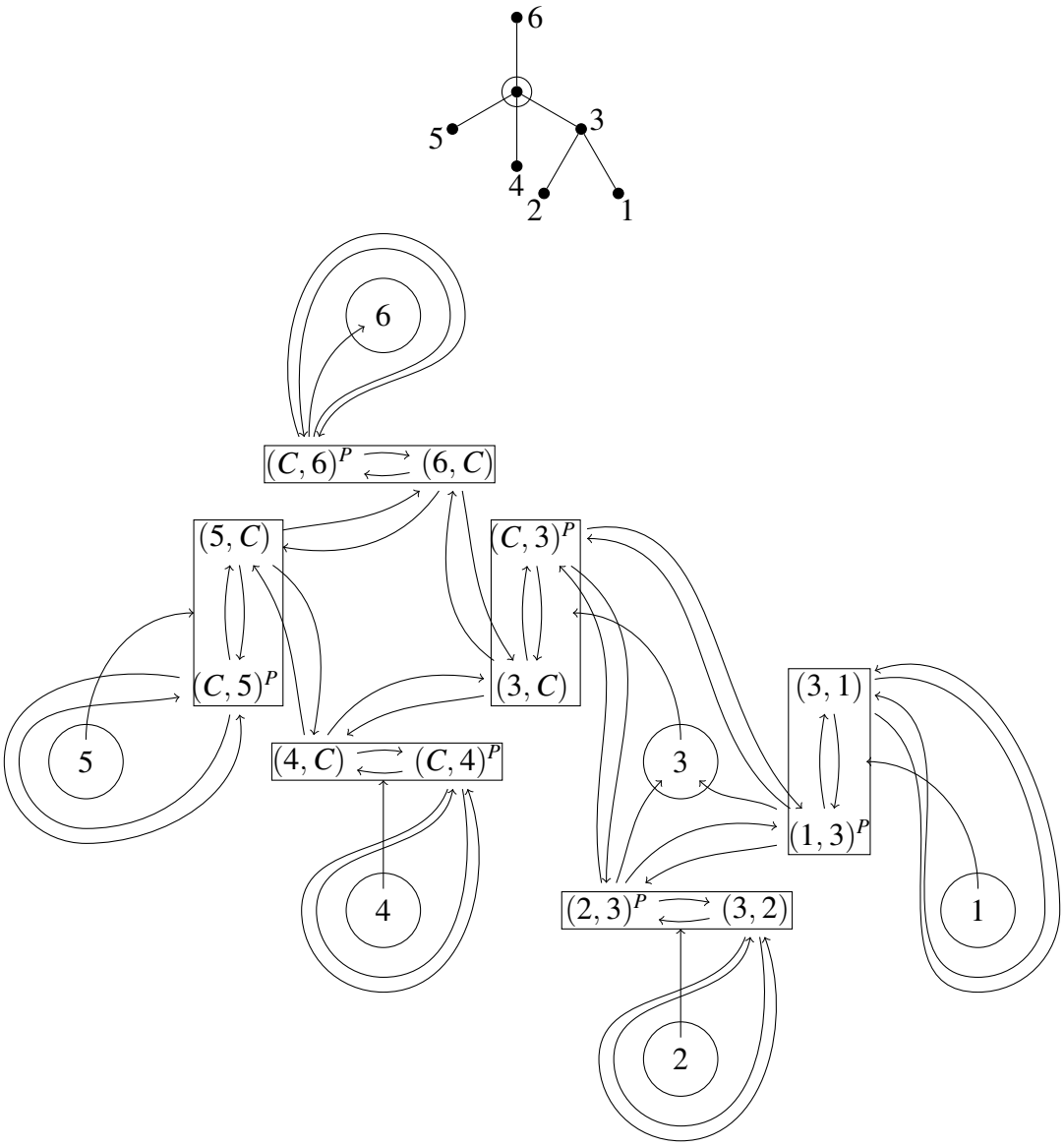

Figure 2.29: A PC-Tree implemented.

way lists are traversed allows the lists that represent C-nodes to be "flipped" and combined in linear-time. The lack of direction in the doubly-linked list relieves the need to reverse elements of a list in order to "flip" the neighbors of a C-node, a quadratic-time operation.

The PC-Tree algorithm is a recursive algorithm. There are four items that get passed to the embedding function at each call:

(i) A two-connected constrained graph G.

(ii) The earliest back vertex i.

(iii) The PC Tree.

(iv) An ordered list of i and all later vertices.

At each step, the algorithm identifies certain vertices and edges that can be embedded trivially. It then reduces the PC-Tree and makes a recursive call to itself. If the graph G is planar, this call returns an embedding of the reduced graph. The algorithm can then add back the trivially embeddable elements and return an embedding of the graph G. If the graph G is non-planar, the recursive call returns a Kuratowski subdivision in the form of a list of branch vertices.

CHAPTER 3

$K_{3,3}$-FREE GRAPHS

This chapter focuses on testing $K_{3,3}$-free graphs for embeddability on the torus. This is a good place to test a novel approach for an algorithm to test toroidality. As expected, a novel approach is not difficult to develop successfully. This chapter presents an adaptation of the PC-Tree planarity algorithm into a practical, polynomial-time algorithm to test whether a $K_{3,3}$-free graph can be embedded on the torus. The algorithm can easily be modified to produce an obstruction to such an embedding (if none exists) or a torus embedding (if one exists).

3.1 – K_5 Subdivisions in a Toroidal Graph

The Kuratowski graphs K_5 and $K_{3,3}$ are both toroidal graphs. The graph K_5 has six different embeddings on the torus, and $K_{3,3}$ has two. These embeddings are shown in Figure 3.1. Because of Kuratowski's Theorem, a non-planar graph that avoids a $K_{3,3}$-subdivision must contain a subdivision of K_5. The ways these subdivisions can overlap and interact in a toroidal graph is critical to understand. First is a simple lemma that will help to identify a $K_{3,3}$ subdivision when adding certain paths to a K_5 subdivision. This lemma has been discovered many times, particularly in [8] and [9].

Lemma 3.1. Let $P_1, P_2, \ldots, P_{10}$ be the ten internally disjoint paths forming a subdivision of K_5, called G. If a new path Q intersects G, and P_i and P_j ($i \neq j$) are two paths intersected by Q consecutively (i.e., Q intersects no other P-paths between P_i and P_j), then there is a subdivision of $K_{3,3}$ in the resulting graph.

Proof. Note that each branch vertex of G belongs to four of the paths $P_1, P_2, \ldots, P_{10}$. Every other vertex belongs to only one path. The proof proceeds by considering the following

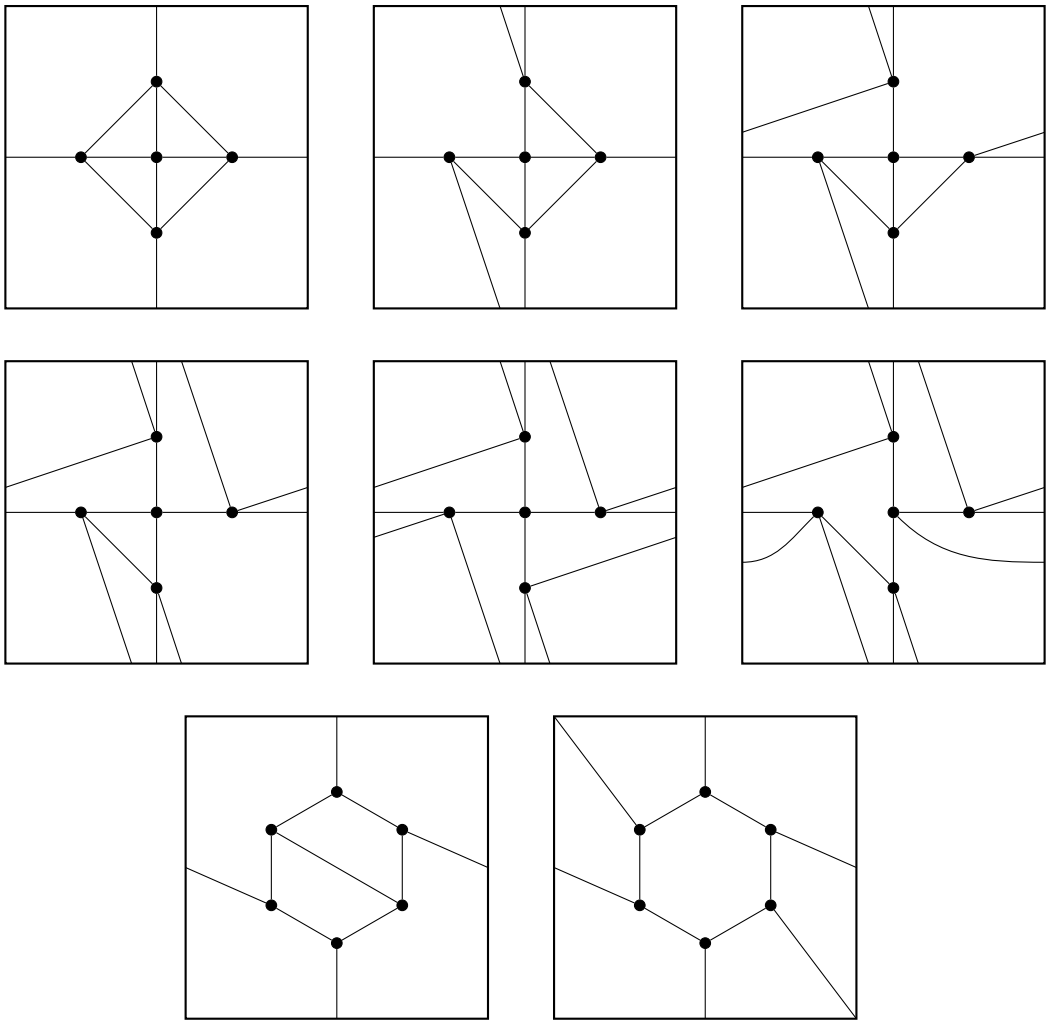

Figure 3.1: The six embeddings of K_5 and the two of $K_{3,3}$ on the torus.

cases:

(i) One of the intersection points is a branch vertex of G.

(ii) Neither intersection point is a branch vertex of G.

(a) P_i and P_j are parallel (i.e., they share no vertices).

(b) P_i and P_j share a common endpoint.

As Figure 3.2 shows, there is a subdivision of $K_{3,3}$ in the resulting graph in each case. The highlighted edge indicates the path Q and the squares and triangles indicate the partite sets of the subdivision of $K_{3,3}$. □

Next is an important theorem of Gagarin, Myrvold, and Chambers [10]. The result com-

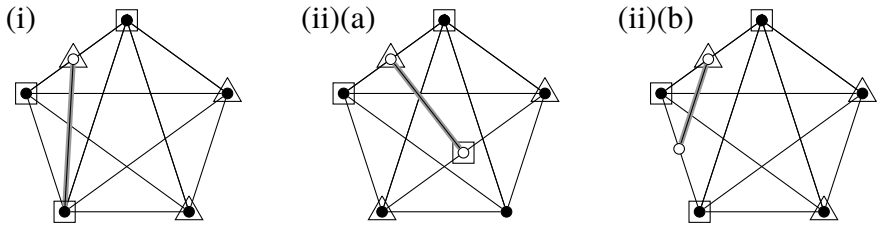

Figure 3.2: A subdivision of $K_{3,3}$ in Case (i), Case (ii)(a), and Case (ii)(b).

pletely characterizes $K_{3,3}$-free toroidal graphs in terms of forbidden subdivisions. This forms the cornerstone for the analysis of K_5 subdivisions in toroidal graphs.

Theorem 3.2. A graph G, with no subdivision of $K_{3,3}$, is toroidal if and only if it avoids subdivisions of the eleven graphs in Figure 3.3.

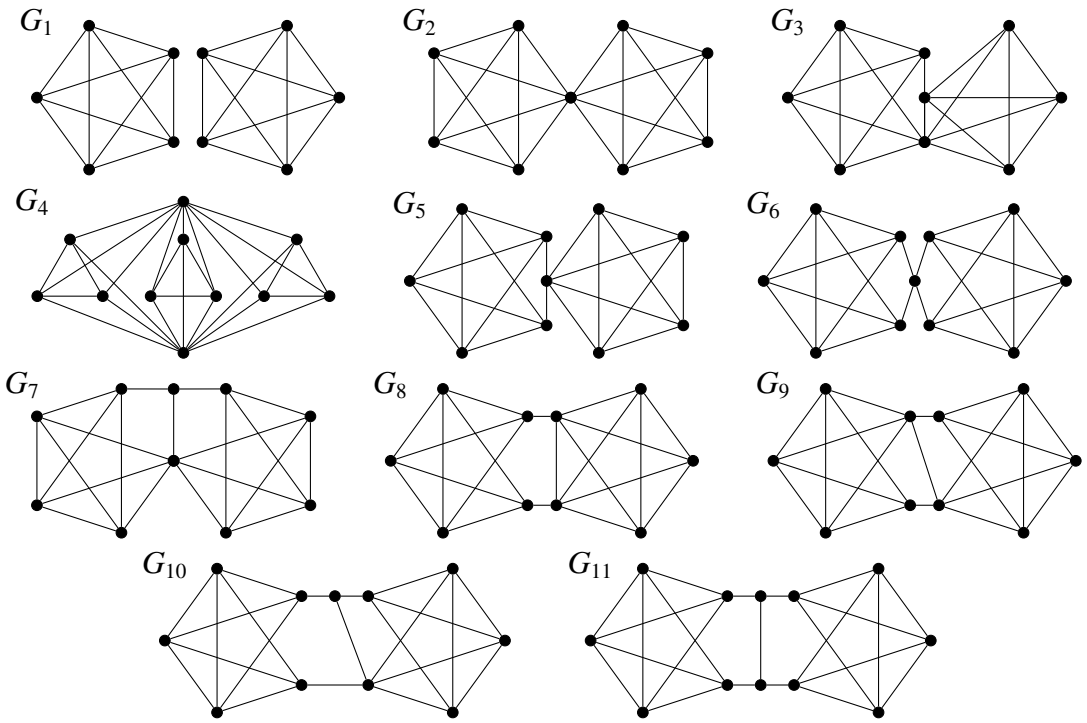

Figure 3.3: The eleven topological obstructions for $K_{3,3}$-free toroidal graphs.

It is important to note some characteristics of these obstructions. First, it is clear that all of them except G_4 are "built" from two subdivisions of K_5. This leads to two obvious questions. First, are there any "subdivision-minimal" combinations of two subdivisions of K_5 missing from this list (i.e., are there any that are toroidal)? Second, what is different about G_4 and

why is it included in this list?

The answer to the first question is "yes": there is one combination of two K_5's that is in fact toroidal, and is therefore not an obstruction. When two subdivisions of K_5 share exactly two branch vertices, the result is the graph M, shown in Figure 3.4, along with its nine embeddings on the torus.

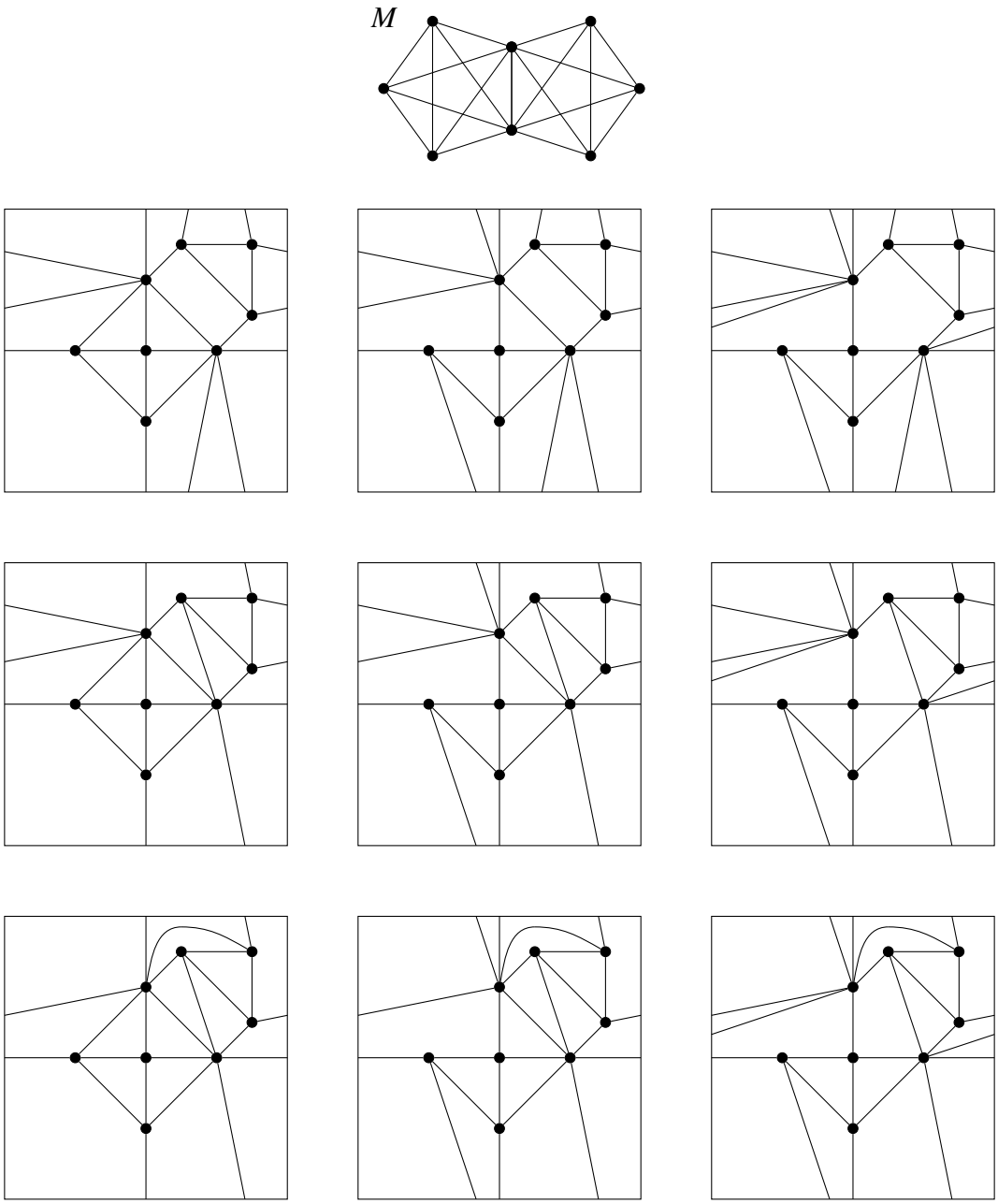

Figure 3.4: The graph M and its nine embeddings on the torus.

See Figure 3.5 to verify that a graph consisting of two subdivisions of K_5 that share three or four branch vertices contains a subdivision of $K_{3,3}$. Therefore further attention is restricted to those graphs in which the two K_5 subdivisions share zero, one, or two branch vertices.

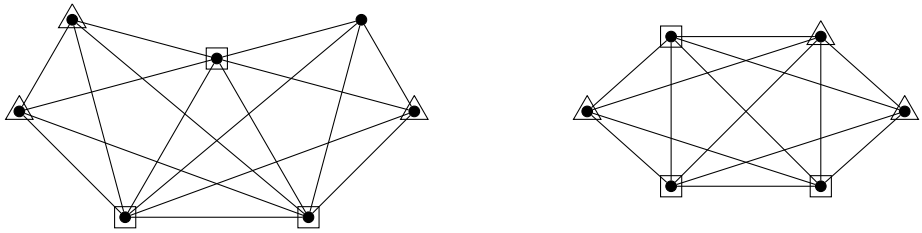

Figure 3.5: Two K_5-subdivisions sharing three or four branch vertices results in a $K_{3,3}$-subdivision (partite sets shown).

Because M is toroidal, it is easier to identify G_4 as three K_5's, each sharing the same two branch vertices (those at the top and bottom in the figure above). Essentially, G_4 is the obstruction that concedes that M is toroidal, but disallows any further K_5's to be added in this fashion.

At this point, there is a small complication in the analysis of how subdivisions of K_5 can interact in a toroidal graph. It is clear that a toroidal graph can contain an *arbitrary* number of K_5 subdivisions: if all the subdivisions have the same branch vertex set, then all the internally disjoint paths running between any two branch vertices can simply be embedded in parallel according to any of the six embeddings of K_5 on the torus. In order to avoid such occurrences where two subdivisions act essentially as one, at least in terms of embedding the graph, we say henceforth that two subdivisions of K_5 are *equivalent* if they have the same set of branch vertices. This qualification leads to the following statement, first discovered by Gagarin and Kocay [9], but reproven here.

Theorem 3.3. If G is a toroidal graph with no subdivision of $K_{3,3}$, then there are at most two non-equivalent subdivisions of K_5 in G.

Proof. Argue by contradiction. Suppose G has three non-equivalent subdivisions of K_5, called H_1, H_2, and H_3. Further, without loss of generality, suppose G is a minor-minimal

such counterexample. Let $v_1, v_2, \ldots, v_5$ be the branch vertices of H_1, and let $P_1, P_2, \ldots, P_{10}$ be the ten internally disjoint paths forming the subdivision of K_5 (some might have only one edge). Now consider $G' = G - \{v_1, v_2, \ldots, v_5\}$. Suppose that the interiors of two paths P_i and P_j are in the same component of G'. Then there is a path Q that connects P_i to P_j that does not pass through a branch vertex of H_1. By Lemma 3.1, there is a subdivision of $K_{3,3}$ in G. Since this contradicts our hypothesis, the interiors of each of $P_1, P_2, \ldots, P_{10}$ must be in different components of G'. See Figure 3.6.

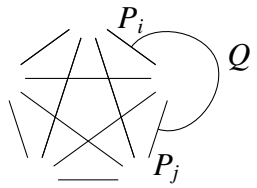

Figure 3.6: If two paths are in the same component in G', then there is a $K_{3,3}$-subdivision in G by Lemma 3.1.

Consider the graph G'' obtained by augmenting each component of G' with copies of the two branch vertices deleted from G incident to that component. The number of components stays the same. Since H_1 and H_2 are different subdivisions of K_5, there is at least one branch vertex y of H_2 that is not a branch vertex of H_1. This vertex must be in a component of G'. If there are branch vertices of H_2 in different augmented components of G'', there would be a path Q from the interior of one path P_i (through y) to a branch vertex of H_1 not on P_i, since H_2 is 5-connected. This creates a subdivision of $K_{3,3}$ by Lemma 3.1, which is a contradiction. Therefore, the branch vertices of H_2 must all be in one augmented component of G''. A similar argument shows that the same is true of the branch vertices of H_3. Since H_1 was chosen without loss of generality, for $i \neq j$, H_i contains no more than two branch vertices of H_j.

If $|V(H_i) \cap V(H_j)| > 2$, then there exists a vertex $u \in V(H_i) \cap V(H_j) \setminus \mathrm{Br}(H_i)$. If possible, choose $u \in \mathrm{Br}(H_j)$. If $u \in \mathrm{Br}(H_j)$, then there is a path from u to a branch vertex of H_i that does not contain any other branch vertex of H_j. If $u \notin \mathrm{Br}(H_j)$, there is still such a path.

If not, the intersected branch vertex of H_j could have been chosen as u. Contract this path away. This destroys no subdivisions of K_5, and the graph is still toroidal. This contradicts the minor-minimality of G. Thus, $|V(H_i) \cap V(H_j)| \leq 2$ for all $i \neq j$.

The next part of the proof establishes that any two subdivisions of K_5, H_i and H_j share exactly two vertices and that these vertices must be branch vertices of both subdivisions.

First, suppose that H_i and H_j share no branch vertices. If, in fact, H_i and H_j share no vertices at all, then G contains the obstruction G_1. If H_i and H_j share a single vertex z, it might be a branch vertex of one or the other, or neither, but not both. If z is a branch vertex of one subdivision, then G contains the obstruction G_5. If z is not a branch vertex of either subdivision, G contains the obstruction G_6. If H_i and H_j share two vertices u and v, there are four cases. If u and v are branch vertices of both subdivisions, G contains the obstruction G_8. If u is a branch vertex of only H_i and v is a branch vertex of only H_j, G contains the obstruction G_9. If u is a branch vertex of one subdivision, and v is a branch vertex of neither, G contains the obstruction G_{10}. If u and v are branch vertices of neither subdivision, G contains the obstruction G_{11}. Now suppose that H_i and H_j share one branch vertex, x. If x is the only vertex shared, G contains the obstruction G_2. If the two subdivisions share a second vertex v, there are two cases. Either v is a branch vertex of one subdivision, in which case G contains the obstruction G_3, or v is a branch vertex of neither, in which case G contains the obstruction G_7. have found the obstruction G_8. Therefore, any two subdivisions of K_5 share exactly two vertices, and those are branch vertices of both subdivisions.

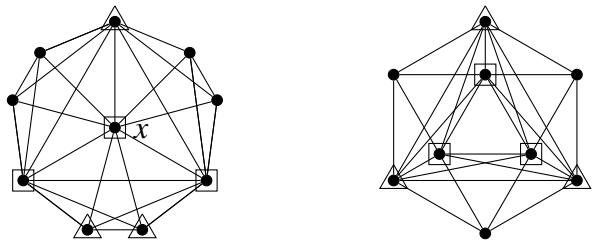

Figure 3.7: The partite sets of the branch vertices of a $K_{3,3}$ subdivision are shown.

If $\bigcap_{i=1}^{3} \mathrm{Br}(H_i) = \{u, v\}$, then G contains the obstruction G_4. If $\bigcap_{i=1}^{3} \mathrm{Br}(H_i) = \{x\}$, then

there is a subdivision of $K_{3,3}$ as shown on the left in Figure 3.7. If $\bigcap_{i=1}^{3} \mathrm{Br}(H_i) = \varnothing$, then there is a subdivision of $K_{3,3}$ as shown on the right in Figure 3.7. □

Corollary 3.4. If G is a toroidal graph with no subdivision of $K_{3,3}$ and exactly two subdivisions of K_5, then G contains a subdivision of M.

Proof. As in the end of the proof of Theorem 3.3, any two subdivisions of K_5 share exactly two vertices, and those are branch vertices of both subdivisions. Such a situation yields exactly a subdivision of M. □

3.2 – A Simple Toroidality Algorithm

Theorem 3.3 leads to a simple, fast, practical toroidality check for $K_{3,3}$-free graphs that works by finding a K_5-subdivision and looking for additional K_5-subdivisions in the side-components. The algorithm presented here uses the PC-Tree algorithm of Shih and Hsu, but does not *extend* it. In fact, this algorithm could use any linear-time planarity check algorithm that provides the branch vertices of a Kuratowski subdivision in a non-planar graph. The algorithm in the next chapter modifies and extends the PC-Tree algorithm, a seemingly necessary step in order to construct a PC-Tree based toroidality algorithm for all graphs. We call this simple algorithm the Side Component algorithm and present it in pseudo-code. See Algorithm 2. When augmenting the components of $G - A$ on line 9 (and again on line 20), Lemma 3.1 implies that each augmented component has exactly two branch vertices of the K_5 subdivision.

This algorithm is linear in the number of vertices. It makes a bounded number of calls to the PC-Tree algorithm, which is a linear-time algorithm. Theorem 3.3 guarantees the PC-Tree algorithm is called at most 21 times: once on G, ten times on the augmented components from deleting the branch vertices of H and ten times on the augmented components from deleting the branch vertices of J. Since this number is bounded and the PC-Tree algorithm itself is linear-time, this algorithm is also linear-time. In the next chapter, knowledge of $K_{3,3}$-free toroidal graphs extends the PC-Tree algorithm into a toroidality algorithm.

Algorithm 2 Side Component Algorithm

```
1:  procedure SideComponent(G)
2:      K5 counter ← 0
3:      if PC-Tree(G) = planar then return toroidal
4:      else if PC-Tree(G) = non-planar (H ≅ K5) then
5:          K5 counter ← K5 counter + 1
6:          A ← set of branch vertices of H
7:          delete A from G
8:          for each component C of G − A do
9:              augment C with copies of the vertices in A adjacent (in G) to vertices of C
10:             if PC-Tree(C) = non-planar (J ≅ K5) then
11:                 K5 counter ← K5 counter + 1
12:                 B ← the set of branch vertices of J
13:         if K5 counter = 1 then return toroidal
14:         else if K5 counter > 2 then return non-toroidal
15:         else (K5 counter = 2)
16:             if |A ∩ B| < 2 then return non-toroidal
17:             else (|A ∩ B| = 2)
18:                 delete B from the component C containing B
19:                 for each component D of C − B do
20:                     augment D with copies of the vertices in B adjacent to vertices of D
21:                     if PC-Tree(D) = non-planar (L ≅ K5) then
22:                         K5 counter ← K5 counter + 1
23:                 if K5 counter = 2 then return toroidal
24:                 else (K5 counter > 2) return non-toroidal
```

CHAPTER 4

PCK-TREE ALGORITHM

4.1 – K-nodes

This chapter discusses the PCK-Tree algorithm, an extension of the PC-Tree algorithm, which determines the toroidality of $K_{3,3}$-free graphs. This algorithm achieves exactly the same result as the algorithm presented at the end of Chapter 3. However, there is much more potential to further extend this algorithm into a toroidality check for more, if not all, graphs.

The PCK-Tree algorithm begins exactly like the PC-Tree algorithm. First find any DFS Tree of the given graph G, and from that, identify the post-ordering of the vertices. Proceed by constructing the PC-Tree as usual. If the given graph is planar, the existing PC-Tree algorithm is sufficient, returning an embedding of G on the plane (which also serves as an embedding of G on the torus). If G is non-planar, then it must contain a subdivision of K_5 since G is $K_{3,3}$-free.

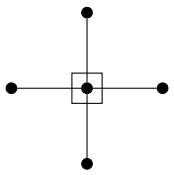

Figure 4.1: A K-node is adjacent to the four known branch vertices of a K_5-subdivision.

The PCK-Tree algorithm extends the PC-Tree algorithm by allowing a new type of vertex, a K-node, that encodes a Kuratowski subdivision. As in Figure 4.1, K-nodes will be represented as a vertex with a square around it. As described in the PC-Tree algorithm, this occurs in the $K_{3,3}$-free case when the algorithm adds the fourth branch vertex of a K_5-subdivision. At this point, the PC-Tree algorithm would simply return the fact that G is

non-planar and indicate the branch vertices of the K_5 (the fifth is found easily). However, the PCK-Tree Algorithm needs to continue since K_5 can be embedded on the torus (in six different ways, in fact). To continue, the algorithm creates a new node called a K-node. Its neighbors are the four known branch vertices of the K_5. As when the algorithm creates a C-node, all full vertices and edges get removed from the tree. Any remaining vertices (together with those added between the fourth and fifth branch vertices) represent the paths from their respective branch vertices to the fifth (as of yet, unadded) branch vertex.

A K_5 subdivision is detected by the PC-Tree algorithm when it encounters a very specific situation. See Figure 4.2 for an example. Note that all the nodes that were full have been embedded (i.e., removed from the tree). The vertices x, y, z, and the newly added vertex w, are the neighbors of the new K-node. They are the four known branch vertices of the K_5 subdivision.

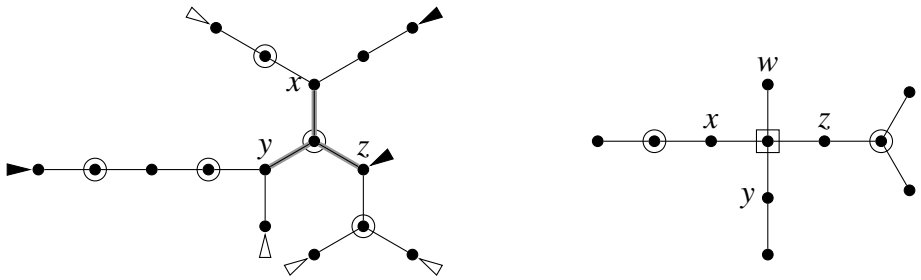

Figure 4.2: Adding vertex w with the terminal edges highlighted and the resulting reduced PCK-Tree.

When the algorithm creates a K-node, it also increments a counter ℓ that keeps track of the number of K_5 subdivisions found so far and provides a list of the known branch vertices of the newest K_5 subdivision. Then the algorithm checks this list against that of any previously identified K_5 subdivision. Call these lists L_1 and L_2. The two K_5's must have exactly two branch vertices in common. If they share fewer than two, the algorithm has found an obstruction and so the graph must be non-toroidal. It is impossible for them to share more than two since this implies a $K_{3,3}$ subdivision. If, at any point, the algorithm creates a third K-node, it has found an obstruction (by Theorem 3.3) and can immediately report that the

graph is non-toroidal. Therefore, it only needs to maintain two lists of branch vertices L_1 and L_2.

It is possible to create two K-nodes in the same step. This seems to be a problem for the graph M, because the algorithm may create two K-nodes, both adjacent to the two shared branch vertices, creating a cycle in the PCK-Tree. However, the DFS numbering prevents this from happening. Such a situation would require a numbering where each TK_5 in M has a non-shared branch vertex *later* than the two shared branch vertices. This implies there is a path from one non-shared branch vertex to the other because the numbering is produced by a DFS Tree of the graph. By Lemma 3.1, the result contains $K_{3,3}$. Therefore, this situation is impossible. See Figure 4.3 for an example numbering.

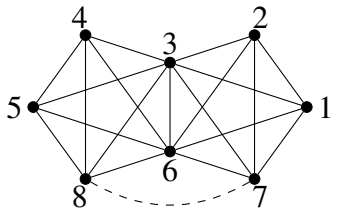

Figure 4.3: Adding vertex 6 in this numbering of M creates two K-nodes in the same step. The dashed edge is necessary for this numbering to be a DFS numbering, but also completes a $K_{3,3}$-subdivision.

Thus, if the PCK-Tree algorithm creates two K-nodes in the same step, it has found two K_5-subdivisions that do not share two branch vertices. Therefore, the procedure need not change in this case. Two lists of branch vertices are made and the same checks above are performed. It happens in this case that the lists will not share exactly two branch vertices; that is, the algorithm will find an obstruction.

When the algorithm adds the fifth branch vertex to any K_5 subdivision, the K-node will be on the terminal path, or it will be full. Following the rules of the PC-Tree algorithm, the PCK-Tree algorithm creates a C-node adjacent to the newly added vertex and the vertices of the terminal path, possibly including the K-node. The edge between the K-node and the C-node can be contracted (the resulting vertex being a C-node) as between two C-nodes in

the PC-Tree algorithm. The K_5 counter has been incremented and the algorithm keeps the list of branch vertices (which can now be completed to include all five) to check against possible future K-nodes. Embedding only the full nodes as always ensures that any empty vertices are still in the PCK-Tree. Therefore, the K-node can be safely removed from the PCK-Tree.

It is possible to add the fifth branch vertex of two K_5-subdivisions at once. In this case, they are sure to share at least one branch vertex (the newly added vertex). The algorithm simply performs the check of the other branch vertices immediately. Both K-nodes are contracted into the new C-node unless the graph is found to be non-toroidal at this step.

The embedding portion of the PCK-Tree algorithm is also similar to that of the PC-Tree algorithm, with a few additions. There are six ways to embed K_5 on the torus, and there are nine ways to embed M on the torus. Based on the value of ℓ, the PCK-Tree algorithm can figure out which embeddings of K_5 it can choose to create an embedding of the graph. If $\ell = 1$, it can choose any of the six because then the side components of the K_5-subdivision are all planar. If $\ell = 2$, there are three it can choose for the first K-node it expands, and based on an embeddings of M, it can finish the embedding of the graph when it expands the second K-node because the side components of the M-subdivision are all planar.

4.2 – Example

For an example, consider the $K_{3,3}$-free graph in Figure 4.4. The steps the PCK-Tree algorithm takes on this graph are described below. Let the counter for K_5-subdivisions be called ℓ (initially 0) and let the lists of branch vertices be L_1 and L_2 (both initially empty).

Until adding vertex 8, the PCK-Tree algorithm works exactly like the PC-Tree algorithm. This results in the PC-Tree shown on the left side of Figure 4.5. When adding vertex 8, there are three terminal edges all incident to a C-node. At this point, the algorithm recognizes a K_5-subdivision with branch vertices 1, 2, 7, and 8. Therefore, it sets $\ell = 1$ and $L_1 = \{1, 2, 7, 8\}$. The K-node created is shown in the center of Figure 4.5.

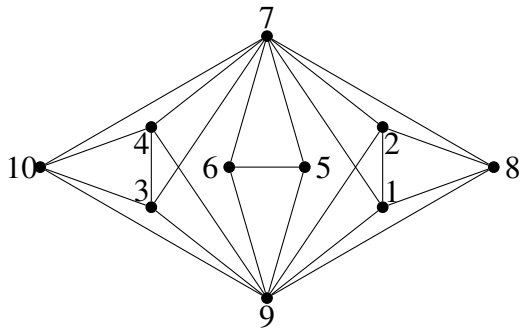

Figure 4.4: A $K_{3,3}$-free graph with a DFS numbering.

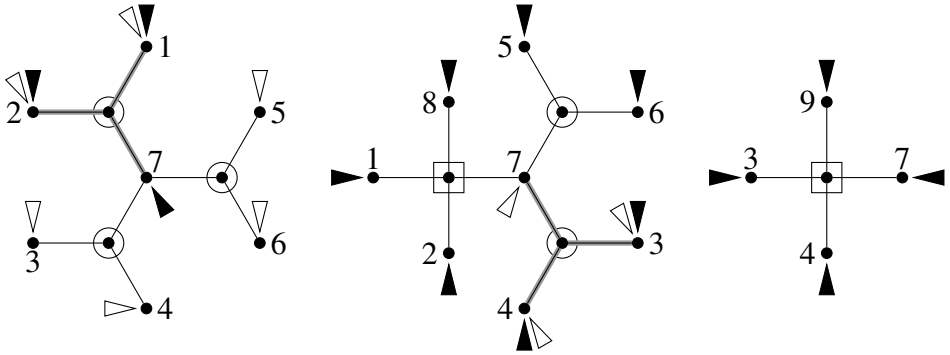

Figure 4.5: Example. The PCK-Tree when adding vertex 8, vertex 9, and vertex 10.

Next, the PCK-Tree algorithm adds vertex 9. There are three terminal edges all incident to a C-node. See the center of Figure 4.5. First, the algorithm completes the first K-node, because it is a full vertex, by setting $L_1 = \{1, 2, 7, 8, 9\}$. Then it recognizes a second K_5-subdivision due to the configuration of terminal edges. The algorithm sets $\ell = 2$ and $L_2 = \{3, 4, 7, 9\}$. Then it checks to make sure the lists of branch vertices are compatible before continuing. L_1 and L_2 currently share exactly two members, so no obstruction is detected. The algorithm continues as usual by embedding all full vertices, thus removing them from the PCK-Tree, and creating a new K-node for the second K_5-subdivision. The result is on the right side of Figure 4.5. In the last step, the algorithm adds vertex 10 which completes the second K_5-subdivision. The list L_2 gets updated to $\{3, 4, 7, 9, 10\}$ and L_1 and L_2 still share only two members. Therefore, the graph is toroidal.

Since there are two K_5 subdivisions in the example graph, there must be an M-subdivision.

The algorithm expands the K-nodes accordingly when stepping back out of the recursive calls of the algorithm. One of the possible embeddings is shown in Figure 4.6.

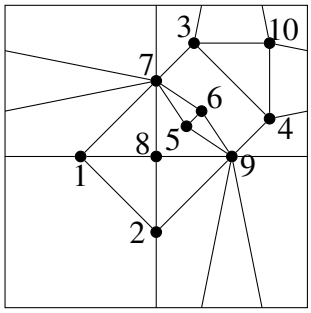

Figure 4.6: Example. A torus embedding of the example graph.

4.3 – Implementation, Complexity, and Correctness

The PCK-Tree algorithm is an extension of the PC-Tree algorithm and borrows its implementation from the planarity check. A K-node is implemented similarly to a C-node: the only necessary addition is a flag in each edge incident to the implicit vertex to indicate that it is a K-node. Besides this extension of the PC-Tree data structure, the algorithm also maintains the aforementioned counter for the number of K_5-subdivisions found in the graph and lists of the known branch vertices of those K_5-subdivisions.

By Theorem 3.3, there are at most two non-equivalent K_5-subdivisions in a toroidal $K_{3,3}$-free graph. Thus, the PCK-Tree algorithm must maintain at most two lists of branch vertices (each of size at most 5), and these need only be checked against one another. These additions are clearly independent of the number of vertices in the graph. Therefore, extending the PC-Tree algorithm to the PCK-Tree algorithm does not increase the complexity. The PC-Tree algorithm is linear-time, and so is the PCK-Tree algorithm.

To close this chapter, a proof of the following result is given.

Theorem 4.1. The PCK-Tree algorithm correctly identifies toroidal and non-toroidal $K_{3,3}$-free graphs, producing an embedding of a toroidal graph, or a subdivision of a toroidal obstruction (from Theorem 3.2) in a non-toroidal graph.

Proof. Let G be a $K_{3,3}$-free graph. If G is toroidal, then it contains at most two non-equivalent K_5-subdivisions by Theorem 3.3 and if it contains two non-equivalent K_5-subdivisions, then it contains a subdivision of M by Corollary 3.4. From the PC-Tree algorithm, the only way to find a K_5 subdivision is to encounter the configuration in Figure 2.12. The PCK-Tree algorithm detects this configuration, extracts the necessary information about the subdivision's branch vertices, and performs a reduction to the PCK-Tree. Because G is toroidal, the counter ℓ remains less than three and the lists L_1 and L_2 (if both exist) agree in exactly two members. In this case, the PCK-Tree algorithm adds all the vertices of G to the PCK-Tree and then steps back out of the recursive calls building a torus embedding of G. This is mostly done according to the PC-Tree algorithm. The only additional method involves the PCK-Tree algorithm choosing one of the six embeddings of K_5 on the torus when expanding a K-node, and ensuring that it is one of the three embeddings compatible with the nine embeddings of M on the torus if $\ell = 2$. Therefore, the PCK-Tree algorithm produces a torus embedding of the graph G.

If G is non-toroidal, then it contains one of the obstructions in Figure 3.3 by Theorem 3.2. These are exactly the graphs that have more than two K_5-subdivisions or have two K_5-subdivisions that share fewer than two branch vertices. Therefore, when the PCK-Tree algorithm finds either the second or third K_5-subdivision, it declares G to be non-toroidal and can provide the appropriate obstruction from Figure 3.3. This detection is outlined in the proof of Theorem 3.3. □

CHAPTER 5

KURATOWSKI COVERINGS

Both the Side-Component algorithm in Chapter 3 and the PCK-Tree algorithm in Chapter 4 test for toroidality by exploiting the structure and interaction of K_5-subdivisions in $K_{3,3}$-free graphs. For general graphs, a toroidality algorithm could exploit the structure and interaction of Kuratowski subdivisions in the input graph. One way to characterize these interactions is to determine if the edges of a graph can be covered with Kuratowski subdivisions. This chapter contains an investigation of such graphs and a proof that, despite considerable evidence and conjectures, a covering by Kuratowski subdivisions is not a property that enables a polynomial-time toroidality check. The main result is that the Kuratowski cover problem is NP-complete (see Theorem 5.10).

A graph G is *coverable* by graphs $H_1, H_2, \ldots, H_n$ if each H_i is isomorphic to a subgraph of G and every edge of G is in at least one H_i. Often, the H_i's are isomorphic to each other, or are taken from a specific set. For example, K_4 is coverable by three K_3's, but not two. See Figure 5.1.

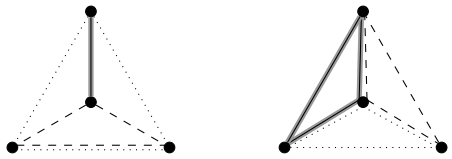

Figure 5.1: K_4 is not coverable by two K_3's (the highlighted edge is not in a K_3), but it is coverable with three K_3's.

5.1 – Covering with Two Kuratowski Subdivisions

Glover conjectured that a graph that is a topological obstruction for a surface of genus g is coverable by $3 - g$ Kuratowski subdivisions. This was disproven by Brunet, Richter,

and Širáň in 1996 [5] . The counterexample they provided (minimally non-toroidal, but not coverable with three Kuratowski subdivisions) is shown in Figure 5.2.

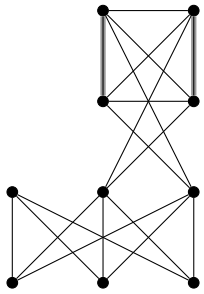

Figure 5.2: A counterexample to Glover's Conjecture.

There is no K_5 subdivision in this graph and no $K_{3,3}$ subdivision contains either of the highlighted edges. It might be expected that a counterexample would be a non-toroidal graph that is coverable with four Kuratowski subdivisions, as opposed to one that is simply not coverable with Kuratowski subdivisions. The graph shown above is "built" from two non-planar graphs. Kuratowski covers seem linked to toroidality and obstructions for embeddability on a surface of genus g. Additionally, the search for a PCK-Tree based toroidality check relies on the structure of Kuratowski subdivisions. Therefore, further investigation of Kuratowski covers is warranted.

Toroidal and $K_{3,3}$-free

The following is a direct result of Theorem 3.3 and Corollary 3.4.

Theorem 5.1. Suppose G is a toroidal graph with no $K_{3,3}$ subdivision that is coverable by two subdivisions of K_5. Assuming the two K_5 subdivisions have different branch sets, then G is a subdivision of M, possibly with a second path between the shared branch vertices.

Proof. By Corollary 3.4, there is a subgraph of G that is a subdivision of M. Each of the covering K_5 subdivisions are needed to cover one side of the M subdivision, with possibly one path between the shared branch vertices left over. Clearly, there can be no other vertices or edges. □

Structure

Let G be a graph. To determine if G can be covered with two Kuratowski subdivisions, consider all possible sets of branch vertices, that is, all $B \subseteq V(G)$ with $5 \leq |B| \leq 12$. (Actually, only vertices with degree 3 or more need be considered.) Robertson and Seymour's solution to the disjoint paths problem [24] provides a polynomial-time algorithm for determining if a given set of vertices is the branch set of a Kuratowski subdivision. This may not provide the Kuratowski subdivision sought for the covering of the graph, but an equivalent subdivision that covers more edges may exist. Using this test rules out sets of vertices that are not branch sets of any Kuratowski subdivision. Let B be the set of vertices chosen to be the branch vertices of either or both of the Kuratowski subdivisions. Let $\overline{B} = V(G) - B$ and let $G[\overline{B}]$ be the graph induced by $\overline{B}$. Since the covering paths cannot intersect themselves or any other path from the same Kuratowski subdivision except at branch vertices, it is clear that all vertices in $\overline{B}$ have degree 2, 3 or 4. If an edge is covered by a path from each Kuratowski subdivision, the edge is called *double-covered*. A configuration of edges and vertices is called *reducible* if it can be replaced with a configuration with fewer edges or vertices such that the graph has the same coverability. Such a replacement process is called a *reduction*.

Lemma 5.2. Here are some observations about the vertices and edges in the induced subgraph $G[\overline{B}]$ of a graph G where B is a set of branch vertices of two Kuratowski subdivisions.

(i) All vertices of degree 2 can be contracted away without changing the coverability of the graph (an elementary reduction).

(ii) If G is coverable by two Kuratowski subdivisions, then a vertex of degree 3 has exactly one incident edge that is double-covered.

(iii) If G is coverable by two Kuratowski subdivisions, then a vertex of degree 4 has no incident edges that are double-covered.

(iv) If G is coverable by two Kuratowski subdivisions, then double-covered edges in $G[\overline{B}]$ have endpoints of degree 3.

(v) A vertex of degree 3 adjacent to three vertices of degree 4 is an uncoverable configuration.

Proof. The proofs of these statements are elementary and are omitted. □

Kuratowski subgraphs in $G[\overline{B}]$

One approach to designing a polynomial-time algorithm to test coverability by two Kuratowski subdivisions is to find a set of unavoidable configurations in $G[\overline{B}]$ and show that they are all reducible. Ideally, once all possible reductions have been performed, the result is a small number of vertices in $\overline{B}$, which could be checked by brute force.

Theorem 5.3. In a reduced graph, $G[\overline{B}]$ contains no subgraphs isomorphic to K_5 or $K_{3,3}$.

Proof. Suppose $G[\overline{B}]$ contains a K_5 subgraph. Since K_5 is 4-regular and there must be paths leading to the vertices in B, there must be a vertex of degree 5 in $\overline{B}$. This is a contradiction.

Suppose $G[\overline{B}]$ contains a $K_{3,3}$ subgraph. Clearly $K_{3,3}$ cannot be covered with one path. It is also clear that since $K_{3,3}$ is 3-regular, a path must enter at a vertex of degree 4 and only one path enters the $K_{3,3}$ at that vertex. So, there are at least two paths, each entering and exiting at a different vertex; there must be at least four vertices of degree 4. There cannot be five vertices of degree 4 and one of degree 3 since a vertex of degree 3 adjacent to three vertices of degree 4 is uncoverable. Thus, there are either four or six vertices of degree 4 in the $K_{3,3}$.

First assume there are six vertices of degree 4. Without loss of generality, there is one path from G_1 and two from G_2. The path from G_1 must touch all the vertices of the $K_{3,3}$, so its entry points must be in different partite sets. The only possibility, up to symmetry, is shown on the left side of Figure 5.3 and it determines the paths from G_2.

Now assume there are four vertices of degree 4. Then there is one path from each of G_1 and G_2. There are two degree 3 vertices that must be adjacent and the shared edge must be double-covered. Both paths must touch all the vertices of the $K_{3,3}$, so their entry points must be in different partite sets. The only possibility, up to symmetry, is shown on the right side of Figure 5.3. In the first case, the $K_{3,3}$ is reducible to two adjacent vertices of degree

4 where the edges leaving one of the vertices correspond to the edges leaving one of the partite sets of the $K_{3,3}$. In the second case, the $K_{3,3}$ is reducible to a single vertex of degree 4. Therefore, any subgraphs isomorphic to $K_{3,3}$ in $G[\overline{B}]$ can be reduced. □

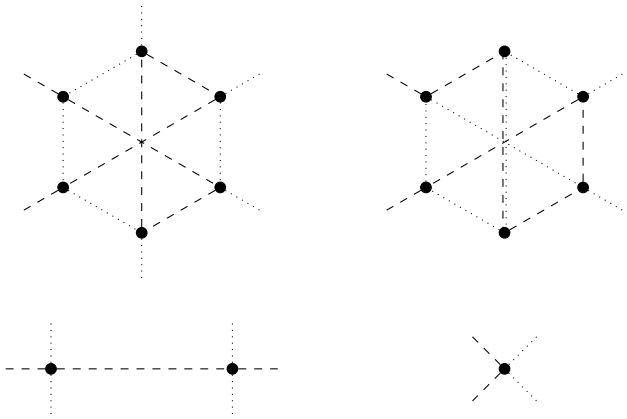

Figure 5.3: Two ways to cover $K_{3,3}$ in $G[\overline{B}]$ and the results after reduction.

5.2 – Using Wagner's Theorem

Later (see Theorem 5.10) we prove that it is NP-complete to detect those graphs whose edges can be covered by two Kuratowski subdivisions. Indeed, we prove that a special case of this problem is NP-complete: it is NP-complete to recognize graphs that avoid a minor of K_5, and can be covered with two $K_{3,3}$ subdivisions. Graphs that have no K_5 minor have been characterized in a Theorem due to Wagner [28] that uses a new concept and a special graph.

A clique sum is a way of combining two graphs into a new graph. In particular, a *k-clique sum* (or, if the meaning is clear, a *k-sum*) of two graphs G and H is defined by identifying the vertices of a K_k subgraph in G with the vertices of a K_k subgraph in H, and then possibly deleting some of the edges of the K_k subgraph in the resulting graph. Therefore, a 1-sum of two graphs is just the operation of identifying one vertex in G with another vertex in H. A 2-sum identifies an edge of G with an edge of H, with the edge optionally being removed. Figure 5.4 shows some clique sums of copies of K_4.

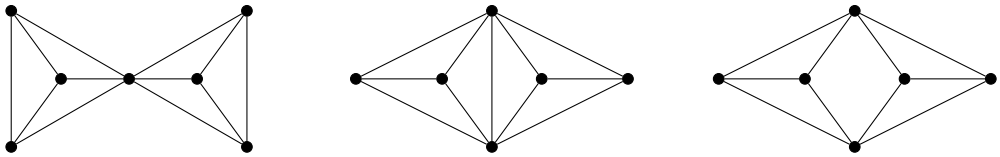

Figure 5.4: A 1-sum and two 2-sums of two K_4's.

Figure 5.5 shows the Möbius Ladder, that is, L_4 with two edges connecting the vertices of degree 2 on opposite corners of the ladder. This graph is called V_8. It is also often described as C_8 with four extra edges, connecting opposite points on the cycle. Finally, V_8 can be viewed as $K_{3,3}$ with two non-incident edges subdivided, and the subdivision vertices adjacent.

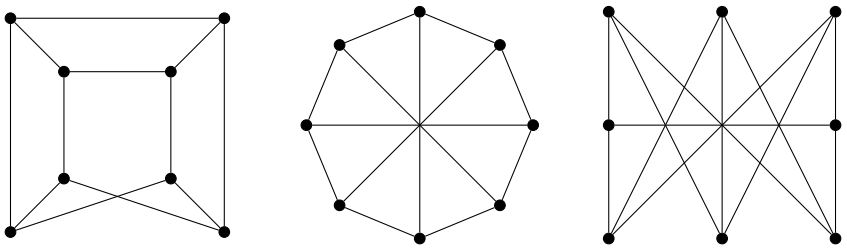

Figure 5.5: Three pictures of V_8: it is the Möbius Ladder, an 8-cycle with chords between opposite vertices, and $K_{3,3}$ with an edge between two subdivided edges.

The following is Wagner's Theorem [28], which characterizes graphs that avoid K_5 as a minor.

Theorem 5.4. *K_5 is not a minor of a graph G if and only if G is made from 0-, 1-, 2-, and 3-sums of planar graphs and V_8.*

For example, $K_{3,3}$ contains no K_5-minor, so it should be the 0-, 1-, 2-, and 3-sum of planar graphs and V_8. There are an insufficient number of vertices in $K_{3,3}$ for V_8 to be used in this sum to produce $K_{3,3}$. $K_{3,3}$ can be built from three copies A, B, and C, of the planar graph K_4 in the following way: Take a 3-sum of A and B, keeping the edges of the triangle. Now take a 3-sum of the result with C on the same triangle, this time deleting the edges of the triangle. The vertices in the triangle are one partite set of the $K_{3,3}$ and the other vertices form the other partite set. See Figure 5.6.

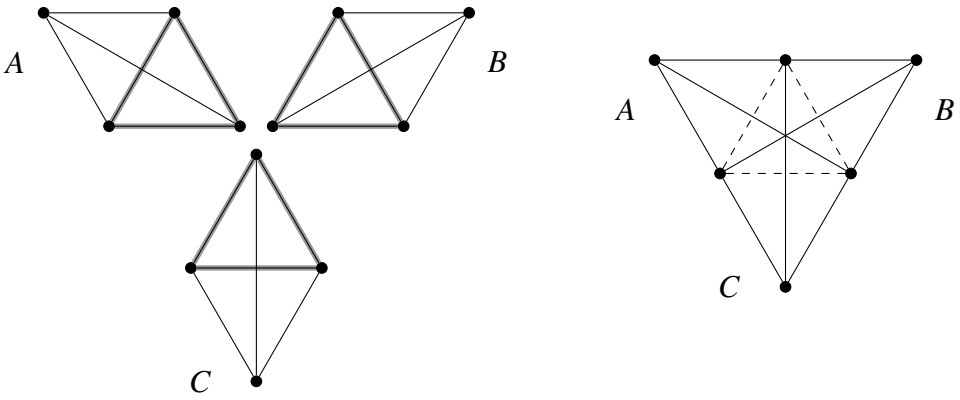

Figure 5.6: $K_{3,3}$ as 3-sums of K_4's. The dashed edges get deleted in the second 3-sum.

As an example of the difficulty in detecting which K_5-minor-free graphs have edge sets that can be covered with two subdivisions of $K_{3,3}$, the following lemma illustrates an analysis for the graphs that are a 2-sum of V_8 with itself.

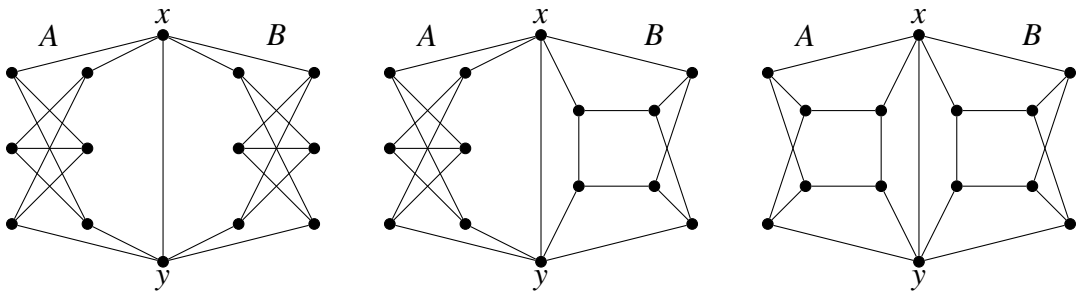

Figure 5.7: Three 2-sums of V_8 with itself.

Proposition 5.5. A 2-sum of V_8 with itself is not coverable with two subdivisions of $K_{3,3}$.

Proof. Let W be a graph that is a 2-sum of two graphs $A, B \cong V_8$. Suppose the 2-cut formed by the 2-sum is $\{x, y\}$. Because V_8 is not edge-transitive, there are three ways to do this. They are shown in Figure 5.7. For sake of contradiction, suppose W is covered with two subdivisions of $K_{3,3}$, one red and one green. Since they have degree 5, neither x nor y are in $W[\overline{B}]$, so each is a branch vertex of at least one $K_{3,3}$ subdivision. Since $\{x, y\}$ is a 2-cut and $K_{3,3}$ is 3-connected, all branch vertices of one $K_{3,3}$ subdivision are in A, or they are all in B. If the red and green branch vertices are both in, without loss of generality, A, then one red xy-path and one green xy-path must cover all of B, including the edge xy. This is clearly

impossible. Suppose that the green branch vertices are in A and the red are in B.

If neither x nor y is a green branch vertex, then the red $K_{3,3}$ subdivision must cover all of B. Since B is isomorphic to V_8, this is impossible. So there is at least one red and at least one green branch vertex in $\{x, y\}$.

Since A is isomorphic to V_8, the green $K_{3,3}$ subdivision cannot cover all the edges of A. Therefore, there is a red edge (and therefore, a red xy-path) in $A - xy$. By the same argument, there must be a green xy-path in $B - xy$. Therefore, the edge xy can be neither green nor red. Thus, W is not coverable by two $K_{3,3}$ subdivisions. □

A 0- or 1-sum of V_8 with itself is not coverable with two $K_{3,3}$ subdivisions because each subdivision would have to cover an entire component or block isomorphic to V_8, but V_8 is not coverable with one subdivision of $K_{3,3}$ (see the right side of Figure 5.5). By Proposition 5.5, a 2-sum of V_8 with itself is not coverable with two $K_{3,3}$ subdivisions. Finally, V_8 has no triangles, so there is no graph that is a 3-sum of V_8 with itself. Therefore, in a graph avoiding K_5 as a minor and coverable by two $K_{3,3}$ subdivisions, there is at most one subgraph isomorphic to V_8.

Further analysis of the structure of K_5-free graphs coverable with two $K_{3,3}$-subdivisions is difficult because it leads to the problem of characterizing graphs coverable with two xy-paths (for two vertices x and y in the graph). This problem is related to finding a Hamiltonian decomposition in the given graph, a concept introduced and used in the next section.

5.3 – Complexity of Covering with Kuratowski Subgraphs

In this section, we prove that recognizing those graphs with edge sets that can be covered by two Kuratowski subdivisions is NP-complete. The proof reduces the well known NP-complete problem of deciding Hamiltonicity of 3-regular graphs to this one.

The Hamiltonian problem is well known and well studied. It asks whether a given graph has a cycle that passes through each vertex exactly once. Such a cycle is called a *Hamiltonian cycle*. Famously, it is NP-complete to determine if a given graph is Hamiltonian. See Figure

5.8.

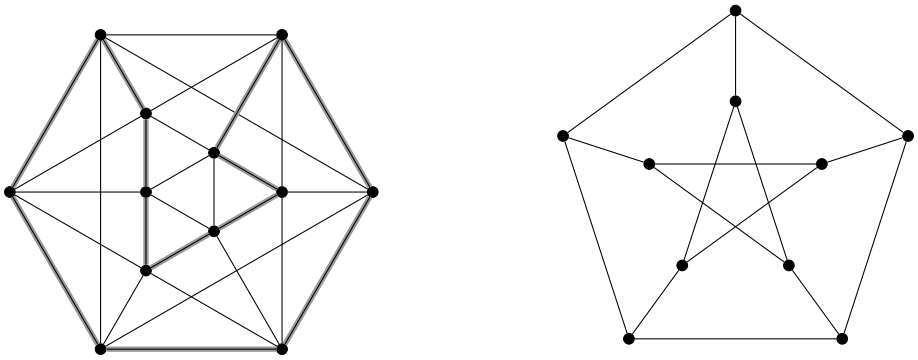

Figure 5.8: The Icosahedron graph is Hamiltonian, but the Petersen graph is not.

A related problem for regular graphs with vertices of even degree is the Hamiltonian decomposition problem. This asks whether the edges of the given $2r$-regular graph can be partitioned into r disjoint cycles, each of which is a Hamiltonian cycle. See Figure 5.9.

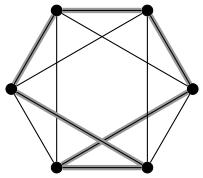

Figure 5.9: A Hamiltonian decomposition. Both the highlighted and unhighlighted edges are Hamiltonian cycles.

Lemma 5.6. The Hamiltonian decomposability problem for 4-regular planar graphs is decidable in at most twice as much time as the Kuratowski coverability problem for $K_{3,3}$-free graphs.

Proof. Let G be a 4-regular planar graph and suppose that the Kuratowski coverability of $K_{3,3}$-free graphs is decidable in polynomial time. Let v be a vertex of G with incident edges $e = uv, f_1 = w_1v, f_2 = w_2v$. Create the graph G' by performing a 2-sum with G and a K_5 at e, deleting the edge e. Then, for each $i = 1, 2$ perform another 2-sum with G' and a K_5 at f_i, deleting the edge f_i, and call the result G'_i. Clearly G'_i is $K_{3,3}$-free. Suppose G'_i is Kuratowski coverable (determinable in polynomial time) for either $i = 1$ or $i = 2$. Then there is a path from u to v and a path from v to w_i that together cover $E(G) - \{e, f_i\}$. Since G

is 4-regular, these paths are edge-disjoint and both are Hamiltonian. It is clear that by adding the edge e to the path from u to v and the edge f_i to the path from v to w_i, two edge-disjoint Hamiltonian cycles in G are created. Thus G is Hamiltonian decomposable. However, if neither G'_1 or G'_2 is Kuratowski coverable, then there are not paths from u to v and from v to w_i that cover $E(G) - \{e, f_i\}$. Therefore, G is not Hamiltonian decomposable. Both G'_1 and G'_2 must be checked because if G is Hamiltonian decomposable, the K_5's must be attached at edges in different Hamiltonian cycles. Although the edges in each Hamiltonian cycle are unknown, either e and f_1 or e and f_2 are in different cycles. Therefore, if G'_i is coverable with two Kuratowski subdivisions, then e and f_i are in different cycles in the Hamiltonian decomposition of G. □

We now introduce three results that, with Lemma 5.6, form a chain that leads to the result in Theorem 5.10. The *line graph* of a graph G is a graph $L(G)$ with $V(L(G)) = E(G)$ and where two vertices of $L(G)$ are adjacent if and only if the corresponding edges in G are incident to the same vertex. First is a theorem due to Sedláček [25].

Theorem 5.7. The line graph of G is planar if and only if

- G is planar,
- $\Delta(G) \leq 4$, and
- for all vertices v of G, if $\deg(v) = 4$ then v is a cut vertex.

Condition (iii) of Theorem 5.7 is irrelevant when the graph is 3-regular as in the following theorem due to Kotzig [18]. This is the link between the Hamiltonian circuit problem and the Hamiltonian decomposition problem.

Theorem 5.8. A 3-regular graph is Hamiltonian if and only if its line graph has a Hamiltonian decomposition.

Finally, the following result states that the Hamiltonian circuit problem is still NP-complete even when several strong restrictions are placed on the graph. This was proven in 1976 by Garey, Johnson, and Tarjan [11].

Theorem 5.9. The Hamiltonian circuit problem is NP-complete even for cubic, planar, 3-

connected graphs.

With this chain of results, the following is an easy consequence.

Theorem 5.10. The Kuratowski Cover Problem for $K_{3,3}$-free graphs is NP-complete.

Proof. Let G be a cubic, planar, 3-connected graph. By Theorem 5.9, it is NP-complete to determine if G is Hamiltonian. Since G is cubic and planar, its line graph $L(G)$ is 4-regular and is also planar by Theorem 5.7. Due to Theorem 5.8, it is also NP-complete to determine if $L(G)$ has a Hamiltonian decomposition. By Lemma 5.6, the Kuratowski Cover Problem for $K_{3,3}$-free has the same complexity as the Hamiltonian decomposition problem for 4-regular planar graphs. Therefore, the Kuratowski Cover Problem for $K_{3,3}$-free graphs is NP-complete. □

REFERENCES

[1] D. Archdeacon, "A Kuratowski theorem for the projective plane," *Journal of Graph Theory*, vol. 5, pp. 243–246, 1981.

[2] 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 System Sciences*, vol. 13, pp. 335–379, 1976.

[3] J. M. Boyer, C. G. Fernandes, A. Noma, and J. C. de Pina, "Correcting and implementing the PC-tree planarity algorithm," 2003.

[4] J. M. Boyer and W. Myrvold, "On the cutting edge: simplified $O(n)$ planarity by edge addition," *Journal of Graph Algorithms and Applications*, vol. 8(3), pp. 241–273, 2004.

[5] R. Brunet, R. B. Richter, and J. Širáň, "Covering genus-reducing edges by Kuratowski subgraphs," *Journal of Graph Theory*, vol. 22(1), pp. 39–45, 1996.

[6] J. Chambers, "Hunting for torus obstructions," *University of Victoria Master's Thesis*, 2002.

[7] R. Diestel, *Graph Theory 4th Electronic Edition*. Springer, 2010, pp. xviii+436.

[8] M. R. Fellows and P. A. Kaschube, "Searching for $K_{3,3}$ in linear time," *Linear and Multilinear Algebra*, vol. 29(3-4), pp. 279–290, 1991.

[9] A. Gagarin and W. Kocay, "Embedding graphs containing K_5-subdivisions," *Ars Combinatoria*, vol. 64, pp. 33–50, 2002.

[10] A. Gagarin, W. Myrvold, and J. Chambers, "The obstructions for toroidal graphs with no $K_{3,3}$'s," *Discrete Math.*, vol. 309(11), pp. 3625–3631, 2009.

[11] M. R. Garey, D. S. Johnson, and R. Tarjan, "The planar hamiltonian circuit problem is NP-complete," *SIAM J. Comput.*, vol. 5(4), pp. 704–714, 1976.

[12] H. Glover, J. Huneke, and C. Wang, "103 graphs that are irreducible for the projective plane," *Journal of Combinatorial Theory Series B*, vol. 27, pp. 332–370, 1979.

[13] J. Hopcroft and R. Tarjan, "Efficient planarity testing," *J. ACM*, vol. 21(4), pp. 549–568, 1974.

[14] W. L. Hsu, "An efficient implementation of the PC-tree algorithm of Shih & Hsu's planarity test," Institute of Informational Science, Academia Sinica, Technical Report TR-IIS-03-015, 2003.

[15] W. L. Hsu and R. M. McConnell, "PQ trees, PC trees, and planar graphs," *Handbook of Data Structures and Applications*, pages, 2001.

[16] C. Jordan, *Cours D'Analyse de l'Ecole Polytechnique, Paris*, second. 1893.

[17] M. Juvan, "Algorithms and obstructions for embedding graphs in the torus," Slovene, *University of Ljubljana, Ph.D. Thesis*, 1995.

[18] A. Kotzig, "From the theory of regular graphs of third and fourth degree," *Casopis pro pestov n matematiky*, vol. 82(1), pp. 76–92, 1957.

[19] K. Kuratowski, "Sur le probl me des courbes gauches en topologie," *Fund. Math.*, vol. 15, pp. 271–283, 1930.

[20] E. Neufeld, "Practical toroidality testing," *University of Victoria Master's Thesis*, 1993.

[21] E. Neufeld and W. Myrvold, "Practical toroidality testing," *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 574–580, 1997.

[22] N. Robertson and P. D. Seymour, "Graph minors IV tree-width and well-quasi-ordering," *J. Combin. Theory Ser. B*, vol. 48(2), pp. 227–254, 1983.

[23] ——, "Graph minors VIII a Kuratowski theorem for general surfaces," *J. Combin. Theory Ser. B*, vol. 52, pp. 153–190, 1990.

[24] ——, "Graph minors XIII the disjoint paths problem," *J. Combin. Theory Ser. B*, vol. 63(1), pp. 65–110, 1995.

[25] J. Sedláček, "Some properties of interchange graphs," *Theory of Graphs and its Applications*, pp. 145–150, 1964.

[26] W. K. Shih and W. L. Hsu, "A new planarity test," *Theoretical Computer Science*, vol. 223, pp. 179–191, 1999.

[27] K. Wagner, "Über eine eigenschaft der ebenen komplexe," *Math. Ann.*, vol. 114, pp. 570–590, 1937.

[28] ——, "Über eine erweiterung des satzes von Kuratowski," *Deutsche Math.*, vol. 2, pp. 280–285, 1937.

[29] J. Woodcock, "A simpler and faster torus embedding algorithm," *University of Victoria Master's Thesis*, 2006.

CURRICULUM VITÆ

Charles J. Suer

5701 Oxford Pl. #746
Louisville, KY 40291
513-544-6923
suerchaj@gmail.com

328 Natural Sciences Building
University of Louisville
Louisville, KY 40292
cjsuer01@louisville.edu

Education

- **University of Louisville** Louisville, KY
  Ph.D. Applied and Industrial Mathematics *(Expected) August 2015*
  - Concentration: Graph Theory
  - Dissertation Topic: An algorithm for embedding graphs on the torus and the role of Kuratowski subdivisions in toroidal graphs.
  - Advisor: André Kézdy
- **University of Louisville** Louisville, KY
  M.A. Mathematics *May 2013*
  - Relevant Courses: Analysis, Algebra, Geometry, Combinatorics and Graph Theory, Mathematical Modeling, Probability Theory and Statistics
- **University of Dayton** Dayton, OH
  B.S. Mathematics (Minors: Computer Science and Theatre) *May 2010*
  - Other Relevant Courses: Differential Equations, Set Theory, Number Theory, Topology, Analysis, Computer Graphics, Operating Systems, Comparative Languages, Data Structures and Algorithms

Teaching Experience

- **University of Louisville** Louisville, KY
  Graduate Teaching Assistant *Aug. 2010 - Aug. 2013*
  - Primary Instructor for the following courses:
    * **Math 111 College Algebra.**
      · linear equalities, inequalities, and their graphs
      · quadratic equations and their graphs
      · advanced properties of polynomial, rational, exponential, and logarithmic functions
    * **Math 190 Precalculus.**
      · polynomial, rational, exponential, logarithmic, and trigonometric functions
      · analytic trigonometry

        * **Math 205 Calculus I.**
            · single-variable calculus: limits, continuity, differentiation, integration, and applications
            · polynomial, trigonometric, exponential, and logarithmic functions
            · some insight into the proofs behind the results
        * **Math 180 Business Calculus.**
            · single-variable calculus: limits, continuity, differentiation, integration, and applications
            · polynomial, exponential, and logarithmic functions
    - Teaching Assistant for the following courses:
        * **Math 105 Contemporary Math**
        * **Math 107 Finite Math**
        * **Math 111 College Algebra**
        * **Math 180 Business Calculus**
- **Private Tutor** Louisville, KY
  High School Math Tutor *Aug. 2012 -*
    - Tutor three local high school students from three different schools
    - Tutor in Algebra, Geometry, Trigonometry, Precalculus, and Calculus.
- **University of Dayton Mathematics Department** Dayton, OH
  Teaching Assistant *Aug. 2007 - May 2010*
    - Teaching assistant for Calculus I, II, and III over six semesters.

Research Experience

- **University of Louisville** Louisville, KY
  Graduate Research Assistant *Aug. 2013 -*
    - Fellowship from the University of Louisville Logistics and Distribution Institute (LoDI).
    - Prepare a poster each year and a presentation each semester on the results of the research.
- **Graduate Research Workshop in Combinatorics** Denver, CO
  Participant *Jul. 2014 - Aug. 2014*
    - Research topics include color-blind index for graphs, acyclic orientation graphs, and generalized Halin graphs.
    - Two weeks of group research with other graduate students and faculty.
- **Bluegrass Summer Research Workshop** Louisville, KY
  Participant, Presenter *Summers 2011 - 2014*
    - Research topics include flooding games on graphs and 2-factors with two cycles.
- **Willamette Valley Mathematics Consortium** McMinnville, OR
  REU Participant *Jun. 2009 - Aug. 2009*
    - Graph Theory Research: variations on the coloring game when played on outer-planar graphs.
    - Prepare a poster, presentation, and paper to highlight the results of the research.

Service and Related Activities

- **University of Dayton Career Seminar** Dayton, OH
  Invited Career Panelist *Oct. 2014*
- **University of Louisville LaTeX Seminar** Louisville, KY
  Presenter *Oct. 2014*
- **Bluegrass Undergraduate Math Symposium** Danville, KY
  Invited Graduate School Panelist *Oct. 2014*
- **University of Dayton Math Club** Dayton, OH
  Secretary (Sep. 2009 - May 2010) *Sep. 2006 - May 2010*
  - Plan events such as Undergraduate Math Day, and the High School Math Contest.
  - Write and grade problems for the High School Math Contest (Mar. 2007 and Mar. 2008)
- **Annual Putnam Competition** Dayton, OH
  Participant *2006, 2007, 2008, 2009*

Awards and Honors

- University of Louisville LoDI Research Fellowship 2013 -
- Great Plains Rocky Mountains Graduate Research Workshop in Combinatorics 2014
- Ken F. and Sandra S. Hohman Graduate Fellowship In Mathematics 2011
- University of Louisville Teaching Assistantship 2010 - 2013
- Undergraduate Math Day: Chair of a Contributed Paper Session 2009
- REU at Linfield College/Willamette Valley Mathematics Consortium 2009
- Award for Excellence in the Sophomore Class 2008
- Dean's List Designee each semester at University of Dayton 2006 - 2010
- Full Tuition Marianist Heritage Award to the University of Dayton 2006 - 2010

Publications

- C. Dunn, J. Firkins Nordstrom, C. Naymie, E. Pitney, W. Sehorn, C. Suer. Clique-Relaxed Graph Coloring. *Involve*, 4(2):127-138, 2011.

Presentations

- The PC-Tree Algorithm, the Torus, and Kuratowski Subdivisions JMM Jan. 2015
- Flooding Games on Graphs BSRW Jul. 2014
- Extending the PC-Tree Algorithm to the Torus (invited) UD Jan. 2014
- Extending the PC-Tree Algorithm to the Torus JMM Jan. 2014
- Packing Trees into Planar Graphs BSRW Jun. 2013
- The Clique-Relaxed Coloring Game BSRW May 2012
- Coloring Games on Outerplanar Graphs JMM Jan. 2010