1-1-2007

On the implementation and refinement of outerplanar graph algorithms.

Tao Deng
University of Windsor

On the Implementation and Refinement of Outerplanar Graph Algorithms

by

Tao Deng

A Thesis
Submitted to the Faculty of Graduate Studies
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2007

Your file Votre référence
ISBN: 978-0-494-34997-7
Our file Notre référence
ISBN: 978-0-494-34997-7

Abstract

An *outerplanar graph* is a graph that can be embedded on the plane such that all the vertices lie on the exterior face and no two edges intersect except possibly at a common end-vertex. Five sequential algorithms had been proposed for recognizing outerplanar graph in the literature and all run in linear time and space. Although among them, the algorithms of Mitchell, Wiegers, and Tsin and Lin are obviously superior, no efforts had been made in comparing their performances during run-time.

In this thesis, the aforementioned three algorithms are implemented and their performances are compared using a large number of randomly generated graphs. Furthermore, the algorithms of Mitchell and Wiegers are modified so that an outerpalnar embedding is generated if the input graph is outerplanar. Correctness proofs of the modification are presented. It is also shown that the complexity of the modified algorithms remain linear in both time and space.

Keywords: *Graph algorithms, outerplanar graph, outerplanar embedding, linear time algorithm, performance evaluation.*

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Tsin whose guidance and encouragement helped me during the research. His attitude of providing only high quality work, has made a deep impression on me. Besides of being an extrordinary supervisor, Dr. Tsin is a dear friend to me. I feel so lucky to get to know Dr. Tsin in my life.

I wish to express my thanks to my thesis committee members, Dr, Wu, Dr. Kao and Dr. Ahmad who used their precious time and provided invaluable suggestions to my thesis.

My parents deserve a special thanks for giving me their unconditional love. I am also very grateful to all my friends for all their help and support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A general definition of *graph* is any mathematical object involving nodes and connections between them. Graph-theoretic problems occur naturally in a great diversity of applications, such as electrical circuits, organic molecules, ecosystems, sociological relationships, databases, and in the flow of control in a computer program.

1.1 Motivation

An outerplanar graph is a graph that can be embedded in the plane so that all the vertices lie on the boundary of the exterior face and no two edges cross each other. Outerplanar graphs appear naturally in a wide variety of applications. For instance, in RNA structure, every secondary structure which consists of a list of base pairs has the structure of an outerplanar graph [52]. In computer networks, message routing is generally an expensive task in terms of time and space complexity. However, for outerplanar network, compact routing schemes [21] and compact fault-tolerant message routing method [21] had been developed. Although in real-life situation, computer networks are usually planar, Frederickson showed that the problem of designing efficient compact routing scheme for planar networks can be reduced to that for a class of outerplaner networks satisfying certain properties [20]. Furthermore, Gonçalves recently showed that every planar graph can be decomposed into two outerplanar subgraphs [25]. The study of outerplanar network thus plays an important role in message routing.

Outerplanar graph has been extensively studied. For instance, while the Hamiltonian cycle problem and the chromatic-number problem are NP-complete and NP-hard, respectively, in general, there exist polynomial-time algorithms for the two problems if the given graph is outerplanar. Mitchell *et al.* [41] showed that the isomorphism problem for maximal outerplanar graphs can be solved in polynomial-time and presented two liner-time algorithms. Bachl et al. [5] showed that the isomorphic subgraphs problem is NP-Complete for outerplanar graphs and is solvable in linear time when restricted to trees. Proskurowski and Sysol [43] presented an efficient algorithm for finding minimum adominating cycle for the biconnected outerplanar graphs. For the problem of list coloring and precoloring extension on the edges of planar graphs, Marx [39] showed that both problems are NP-Complete for bipartite outerplanar graphs.

It is of both theoretical and practical interest to determine if a graph is outerplanar and produce an outerplanar embedding of it if it is. Efficient algorithms had been proposed for this problem on various computer models.

For the parallel model, Diks, Hagerup and Rytter [16] presented an algorithm that runs in $O(\log n \log\log n)$ time using $n/(\log n \log\log n)$ processors on the CREW (*concurrent-read-exclusive-write*) PRAM (Parallel RAM), where n is the number of vertices in the given graph. If the graph is outerplanar and biconnected, then a Hamiltonian cycle will also be produced.

For the distributed model, Kazmierczak and Radhakrishnan [33] presented an asynchronous distributed algorithm that uses $O(n)$ time and transmits $O(m)$ messages to determine if a biconnected network with n-node and m-link is outerplanar.

For the external memory model, Maheshwari and Zeh [37] presented an algorithm that performs $sort(n)$ I/O operations to determine if a biconnected graph is outerplanar, where $sort(n)$ is the number of I/O operations performed to sort a list of n elements.

For the sequential model, a number of linear time and space algorithms for recognizing outerplanar graph had been published. Brehaut proposed the first two algorithms [7]. Both algorithms rely heavily on the planarity testing algorithm of Hopcropt and Tarjan [30] and are thus quite complicated. In the first algorithm, the planarity testing algorithm is first used to assure that the given graph is a planar graph. After that, a dependency subgraph is generated. A coloring is then performed on the dependency subgraph to generate an outerplanar embedding of the given graph. In the second algorithm, a depth-first search is first performed over the given graph to convert the graph into a palm tree [48]. An acceptable adjacency list structure for the palm tree is then generated. A second depth-first search is then performed over the palm tree to produce an ear-decomposition of the graph. Those ears that have more than one edges are then used to form a Hamiltonian cycle of the given graph. Based on the Hamiltonian cycle, the original adjacency structure is modified and a third depth-first search is performed, generating another palm tree and another acceptable adjacency structure. A directed Hamiltonian cycle of the given graph with diagonals is then generated. The given graph is outerplanar if and only if no two diagonals cross each other.

Syslo and Iri [47] presented another depth-first search based algorithm for recognizing outerplanar graphs. Their algorithm uses the fact that a biconnected graph is outerplanar if and only if it is a cycle or it can be reduced to a cycle by repeatedly replacing maximal paths whose internal vertices are of degree two with a single edge. Although this algorithm is simpler than that of Brehaut, it is still quite complicated as it makes multiple passes over the given graph and uses sorting.

Mitchell [41] presented another algorithm which does not use depth-first search. Instead it is based on maximal outerplanar graph - an outerplanar graph such that adding any edge between any two non-adjacent vertices results in a non-outerplanar graph. The idea underlying their algorithm is to transform a given biconnected graph into a maximal outerplanar graph by repeatedly adding edges between non-adjacent vertices. It had been shown that a biconnected graph is outerplanar if and only if it can be transformed into a maximal outerplanar graph.

Wiegers [53] presented yet another algorithm that does not use depth-first search. The algorithm uses an edge coloring technique and repeatedly deletes vertices of degree two or less. It can work directly on graphs that are not biconnected.

Recently, Tsin and Lin [50] presented yet another depth-first search based algorithm for testing and embedding outerplanar graphs. Their algorithm is based on a new characterization theorem of outerplanar graph whose conditions can be efficiently tested during the depth-first search.

So far, no work had been done on comparing the performances of the aforementioned sequential algorithms. Therefore, in this thesis, we shall implement the algorithms and compare their performance based on randomly generated graphs. However, after a preliminary study of the six algorithms, we noticed that the algorithms of Brehaut and that of Syslo *et al.* are clearly inferior to the rest. We shall thus implement and compare the last three algorithms only.

We had also noticed that the algorithms of Mitchell and Wiegers only test for outerplanarity of the given graph. They do not produce an embedding if the graph is indeed outerplanar. We shall thus refine the two algorithms to include such functionality.

1.2 Thesis Statement

In this thesis, a detailed comparison of the algorithms of Mitchell, Wiegers and Tsin's outerplanar graph algorithms will be presented. Firstly, crucial details that were omitted in the original presentation of Mitchell's and Wiegers' algorithm will be filled in. The three algorithms are then implemented and their performances are compared based on a large number of experimental graphs. The graphs are generated randomly and are of different types with different sizes.

While Tsin's algorithm also generates an embedding of the graph if it is indeed outerplanar, Mitchell's and Wiegers' do not. In this thesis, the algorithm of Mitchell and Wiegers, respectively, are modified so that an outerplanar embedding is generated if the input graph is outerplanar. Correctness proofs of the modification are presented. It is also shown that the complexity of the modified algorithms remain linear in both time and space.

1.3 Organizations of Thesis

This thesis is organized into eight chapters. Chapter 1 gives the motivation of the thesis. Chapter 2 introduces the background knowledge of graph theory, graph algorithm, depth-first search and bucket sort. Chapters 3,4 and 5 explain Mitchell's, Wiegers', Tsin and Lin's outerplanar outerplanar graph algorithm, respectively, and present efficient implementation for each of them. Chapter 6 presents and discusses the experimental results. Chapter 7 presents outerplanar embedding algorithms for Mitchell's and Wiegers' algorithm. Chapter 8 is the conclusion.

Chapter 2

Background

2.1 Basic Definition

A graph $G = (V, E)$ consists of two sets V and E.

- The elements of V are called vertices (or nodes).
- The elements of E are called edges
- Each edge is associated with two vertices (possibly identical) called its end-points.

The sets V and E are usually finite. $|V|$ is the ***order*** (the number of vertices) and $|E|$ is the ***size*** (number of edges) of the graph. In an ***undirected*** graph, each edge is associated with an unordered pair (see Figure 2.1) whereas in a ***directed*** graph, each edge is an ordered pair (see Figure 2.2). In this thesis, (u, v) represents an unordered pair, whereas $< u, v >$ represents an ordered pair. If an edge e is associated with an unordered (ordered, respectively) pair (u, v) ($< u, v >$, respectively), we shall write $e = (u, v)$ ($e =< u, v >$, respectively). A direct edge $e =< x, y >$ is considered to be directed from x to y; x is called the ***tail*** and y is called the ***head*** of the edge.

2.1.1 Related Concepts

In this thesis, we shall focus on undirected graph. The following definitions are thus given to undirected graph although they can be easily extended to directed graph.

Definition 1. *A vertex u is* ***adjacent*** *to a vertex v if they is an edge $e = (u, v)$. The two vertices are said to be* ***joint*** *by the edge e.*

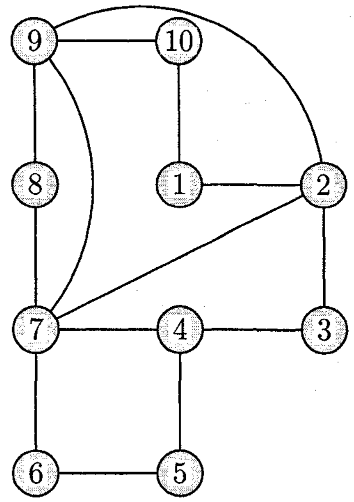


Figure 2.1: an undirect graph

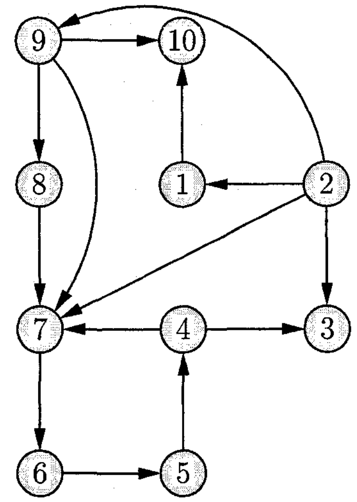


Figure 2.2: a direct graph

Definition 2. *If vertex v is an endpoint of edge e, then v is said to be* ***incident*** *on e, and e is* ***incident*** *on v.*

Definition 3. *Two adjacent vertices are called* ***neighbors****.*

Definition 4. *A* ***self-loop*** *is an edge whose two end-points are identical.*

Definition 5. *A* ***multi-edge*** *is a collection of two or more edges having identical end-points.*

Definition 6. *A* ***proper edge*** *is an edge that joins two distinct vertices.*

Definition 7. *A* ***simple graph*** *is a graph that has no self-loops or multi-edges.*

Definition 8. *The* ***degree*** *of a vertex v (denoted by* ***Deg(v)****) in a graph G, is the number of proper edges incident on v plus twice the number of self-loops.*

Definition 9. *A* ***path*** *in a graph is a sequence of vertices such that from each vertex there is an edge to the next vertex in the sequence. The first vertex is called the* ***start vertex*** *and the last vertex is called the* ***end vertex****. Both of them are called* ***end or terminal vertices*** *of the path. The other vertices in the path are* ***internal vertices****.*

Definition 10. *A* ***cycle*** *is a path such that the start vertex and end vertex are the same.*

Definition 11. *A graph is* ***connected*** *if between every pair of vertices there is a path.*

Definition 12. *A* ***subgraph*** *of a graph G is a graph whose vertex and edge sets are subsets of those of G. A* ***spanning subgraph*** *of G is a subgraph of G whose vertex set is same as that of G.*

Definition 13. *A **connected component** of a graph G is a connected subgraph H such that no subgraph of G that properly contains H is connected.*

Definition 14. *A simple graph $G = (V, E)$ is **isomorphic to** a simple graph $H = (V', E')$ if there exists a bijection $f : V \rightarrow V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$*

Definition 15. *A **cut-vertex** is a vertex whose removal increases the number of connected components.*

Definition 16. *A **biconnected graph** is a graph without cut-vertex.*

Definition 17. *A **cut-edge** (also known as **bridge**) is an edge whose removal increases the number of connected components.*

Definition 18. *A **tree** is a connected graph with no cycles.*

Definition 19. *A **spanning tree** of a graph G is a spanning subgraph of G that is a tree (see Figure 2.3).*

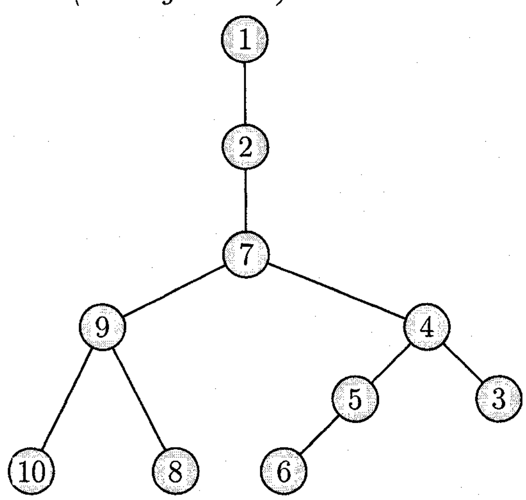


Figure 2.3: A spanning tree of the graph in Figure 2.1

Definition 20. *A simple graph is a **complete graph** if every pair of vertices is joined by an edge. The complete graph with n vertices is denoted by K_n (see Figure 2.4, 2.4).*

Definition 21. *A **2-vertex** is a vertex of degree 2 and whose neighbors are adjacent.*

Definition 22. *A simple graph is **bipartite** if its vertices can be partitioned into two disjoint sets (called **partite sets**) in such a way that no edge joins two vertices in the same set.*

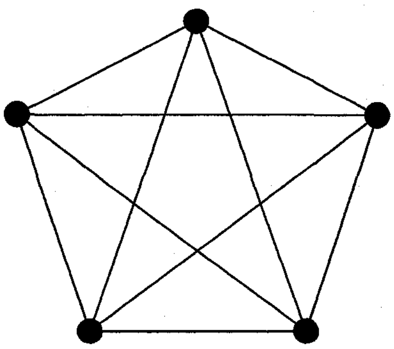


Figure 2.4: K_5

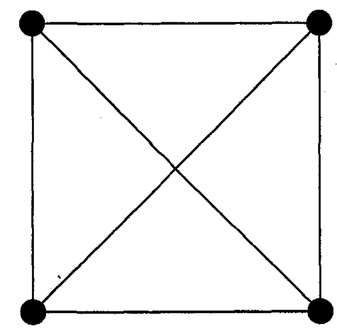


Figure 2.5: K_4

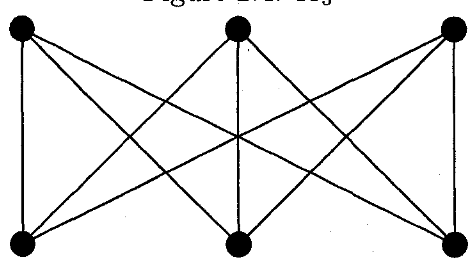


Figure 2.6: $K_{3,3}$

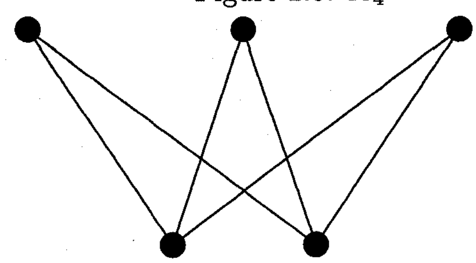


Figure 2.7: $K_{2,3}$

Definition 23. *A **complete bipartite graph** is a simple bipartite graph in which each vertex in one partite set is adjacent to all the vertices in the other partite set. If the two partite sets have cardinalities r and s, then this graph is denoted by $K_{r,s}$ (see Figure 2.6, 2.7).*

Definition 24. *A graph is **Hamiltonian** if it has a spanning cycle.*

Definition 25. *Two graphs G and H are **homeomorphic** if both of them can be obtained from the same graph by replacing edges with paths.*

Definition 26. *A **planar embedding** of a graph is a graphical representation of the graph on the plane (with* dots *representing vertices and* line segment *joining two dots representing edges joining the two corresponding vertices) such that no two edges intersect except at an end-point. The edges partition the plane into regions, called **faces**. The edges surrounding a region is called the **boundary** of that region. There is exactly one face with unbound area called the **exterior face**.*

Definition 27. *A graph is **planar** if it has a planar embedding in the plane.*

Definition 28. *A graph is called **outerplanar** if it has an embedding in the plane such that all the vertices lie on the boundary of the exterior face.*

Definition 29. *A **maximal outerplanar graph** is an outerplanar graph such that adding an edge to join any two non-adjacent vertices results in a non-outerplanar graph.*

Definition 30. *An* ***outer edge*** *is an edge which lies on the boundary of the exterior face.*

Definition 31. *The* ***inner edge*** *is an edge which does not lie on the boundary of the exterior face.*

2.2 Representation of Graph

2.2.1 Adjacency Matrix

An ***adjacency matrix*** of a graph $G = (V, E)$ is an $|V| \times |V|$ matrix M, such that $M[i, j] = 1$ if and only if vertex v_i and vertex v_j are adjacent. Adjacency matrix is the simplest way to represent graphs. However, the time and space complexity are $\Omega(|V|^2)$ as it requires $\Theta(|V|^2)$ memory locations to store the matrix M and $\Theta(|V|^2)$ time to initiate the matrix. Figure 2.1 is an adjacency matrix for the graph in Figure 2.1.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
v_1	0	1	0	0	0	0	0	0	0	1
v_2	1	0	1	0	0	0	1	0	1	0
v_3	0	1	0	1	0	0	0	0	0	0
v_4	0	0	1	0	1	0	1	0	0	0
v_5	0	0	0	1	0	1	0	0	0	0
v_6	0	0	0	0	1	0	1	0	0	0
v_7	0	1	0	1	0	1	0	1	1	0
v_8	0	0	0	0	0	0	1	0	1	0
v_9	0	1	0	0	0	0	1	1	0	1
v_{10}	1	0	0	0	0	0	0	0	1	0

Table 2.1: Adjacency matrix of the graph in Figure 2.1

2.2.2 Adjacency List

An ***adjacency list*** of a graph $G = (V, E)$ consists of an $|V|-$element array of pointers, where the ith element points to a linked list of the vertices adjacent to the vertex v_i. without loss of generality, we shall use v_i and i interchangeably. $AList(i)$ denotes the adjacency list of vertex i. $j \in AList(i)$ implies that vertex j is adjacent to vertex i. To initialize the adjacency list, $O(|E|)$ time is sufficient. We shall use adjacency lists to represent the given graph in this thesis.

$1 \longrightarrow 2 \longrightarrow 10$
$2 \longrightarrow 1 \longrightarrow 3 \longrightarrow 7 \longrightarrow 9$
$3 \longrightarrow 2 \longrightarrow 4$
$4 \longrightarrow 3 \longrightarrow 5 \longrightarrow 7$
$5 \longrightarrow 4 \longrightarrow 6$
$6 \longrightarrow 5 \longrightarrow 7$
$7 \longrightarrow 2 \longrightarrow 4 \longrightarrow 6 \longrightarrow 8 \longrightarrow 9$
$8 \longrightarrow 9 \longrightarrow 7$
$9 \longrightarrow 2 \longrightarrow 8 \longrightarrow 7 \longrightarrow 10$
$10 \longrightarrow 1 \longrightarrow 9$

Table 2.2: Adjacency lists of the graph in Figure 2.1

Definition 32. ***Cross-pointer linked lists*** *are the adjacency lists of the graph* $G = (V, E)$ *such that for each vertex* v *in* $AList(u), u \in V$, *there is cross-pointer between the vertex* v *in* $AList(u)$ *and the vertex* u *in* $AList(v)$.

2.3 Graph Traversing Techniques

A search algorithm takes a problem as input, evaluates a number of possible solutions, and returns a solution to the problem. The set of all possible solutions to a problem is called the *search space.*

Among all the search algorithms, tree search algorithm is the heart of all search techniques, and is one of the central algorithms of many game playing programs. A tree traversal is a process of visiting each vertex in a tree data structure. Such traversal can be classified by the order in which the nodes are visited. For instance, level by level (Breadth-first search), reaching a leaf vertex first before backtracking (Depth-first search), alternative-deepening search, depth-limited search, bidirectional search and uniform-cost search.

2.3.1 Depth First Search

Depth First Search (abbreviated as DFS), as its name implies, is a graph-search method that searches "deeper" when possible. Specifically, a DFS extends the current path as far as possible before backtracking to the last reached vertex and trying the next alternative path.

DFS was first used by Tarjan in his algorithms for finding biconnected component and strongly connected component [48]. Later, Tarjan and Hopcroft used it to develop a linear-time algorithm for recognizing planar graph [30]. Since then, depth-first search has been used in developing optimal algorithm for a vast variety of graph-theoretic problems.

Owing to the success in using depth-first search to develop efficient graph algorithms on the sequential computers, researchers in parallel computation had attempted to adapt the technique to parallel computers. Unfortunately, very few progresses were reported. Finally, Reif proved that depth-first search is an inherently sequential technique [44].

It turned out that depth-first search is much more adaptable to the distributed processing setting. Chueng [9] presented the first depth-first search algorithm that runs on an asynchronous computer network. The algorithm takes $2m$ time and transmits $2m$ messages each with $O(1)$ length, where m is the number of links in the network. Awerbuch [4] improved the time bound to $4n$, where n is the number of nodes in the network (note that $m = O(n^2)$). Lakshmanan *et al.* [35] tightened the time bound to $2n - 2$. Cidon [11] showed that the message bound can be reduced to $3m$; however, Tsin [49] later showed that Cidon's algorithm does not always perform a depth-first search over the network correctly. Tsin then corrected the flaws in Cidon's algorithm and showed that the time and message complexity of the corrected algorithm are actually same as those of Lakshmanan *et al.* Tsin further showed that by extending the message length from $O(1)$ to $O(\log n)$, the time complexity of the corrected Cidon's algorithm can be improved to $n(1 + r)$, where $0 \leq r < 1$. Sharma *et al.* [34,45] showed that one can trade message size for time and message by using messages of length $O(n)$ to reduce the time and message to $2n - 2$. Makki *et al.* [38] improved the bounds to $n(1 + r)$, where $0 \leq r < 1$ by using the dynamic backtracking technique. Recently, Turau [51] showed that depth-first search is also adaptable to wireless sensor network.

On the *external-memory* model (a model in which the input size is larger than the internal memory size), Chiang *et al.* [10] proposed a depth-first search algorithm that requires $O(\lceil n/M \rceil scan(m) + n)$ I/O operations, where M is the size of the internal memory, n and m are the number of vertices and the number of edges, respectively, of the given graph, and $scan(m)$ is a primitive which is the number of I/O operations needed to *read* m items striped across the external disks that form the external memory. Buchsbaum *et al.* [1] introduced the *buffered reposi-*

tory tree and used it to develop another depth-first search algorithm that requires $O((n + m/B)\log_2(n/B) + sort(m))$ I/O operations, where B is the number of items an I/O operation can transfer from/to an external disk and $sort(m)$ is another primitive which is the number of I/O operations needed to *sort* m items striped across the external disks. The algorithm outperforms that of Chiang *et al.* when $M = o((n/B)/\log_2(n/B))$. For planar graph, Arge *et al.* [2] presented a depth-first search algorithm that requires $O(sort(n)\log(n/m))$ I/O operations.

The following is a brief description of depth-first search:

Initially, all the edges in the graph $G = (V, E)$ are unexplored and all vertices are unvisited. An arbitrary vertex r is chosen as the starting point of the depth-first search. Vertex r thus becomes the *current vertex* of the search. In general, let v be the current vertex of the search. An unexplored edge incident on v is chosen. If the edge does not lead to an unvisited vertex, it is discarded and another unexplored edge is chosen. This step is repeated until either an unexplored edge whose other end-point w is unvisited is encountered or vertex v runs out of unexplored edge. In the former case, the search advances to vertex w making it the current vertex. In the latter case, the search backtracks to the vertex u from which v was discovered as an unvisited vertex earlier.

A depth-first search creates a spanning tree, called ***depth-first search spanning tree*** (abbreviated as DFS-tree), of the given graph. The spanning tree consists of all those edges the search uses to advance from a current vertex to an unvisited vertex. An edge in the graph is called a ***tree edge*** if it belongs to the DFS-tree and is called a ***back edge***, otherwise. Let $e = (u, v)$ be a tree edge. Vertex u is the ***parent*** of vertex v if vertex u is visited before vertex v during the search. Vertex v is called a ***child*** of vertex u.

The depth-first search also labels each vertex v with an integer, called the ***depth-first search number*** of v, which shall be denoted by $\boldsymbol{dfs(v)}$. The integer is the rank of vertex v in the ordering the vertices are visited by the depth-first search. Specifically, $dfs(v) = k$ if vertex v is the k^{th} unvisited vertex being turned into a current vertex by the search.

The following is a formal description of depth-first search.

Algorithm 1 DFS(v, u)

Input: The adjacency lists of $G = (V, E)$;
{**comment:** vertex u is the parent of vertex v}
$dfs(v) \leftarrow count$; $count \leftarrow count + 1$; **comment:** /* $count$ is initialized to 1 */
for each w in the adjacency list of v **do**
  if w is unvisited **then**
    DFS(w, v)
  end if
end for

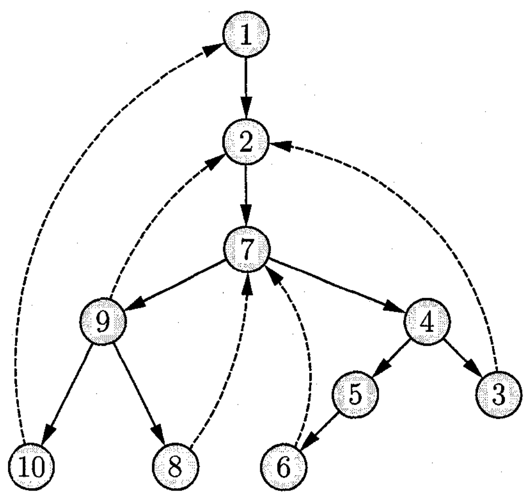


Figure 2.8: a DFS spanning tree of the graph in Figure 2.1

2.4 Planar Graphs and Outerplanar Graphs

2.4.1 Planar Graph

Planar graph arises naturally in real-life situation. For instance, railway maps, electric circuits are planar graphs.

Kuratowski gave the first characterization theorem for planar graphs, now known as the Kuratowski's theorem.

Theorem 1. *An undirected graph is planar if and only if it does not contain a subgraph that is homeomorphic to K_5 or $K_{3,3}$.*

Unfortunately, there is no apparent way of using Kuratowski's theorem to produce an efficient algorithm for planarity testing. Auslander and Parter [3] presented the first planarity algorithm. The algorithm runs in $O(n^3)$ time, where n

is the number of vertices in the graph. Later, Goldstein [23] spotted an error in Auslander and Parter's algorithm and corrected it.

The first linear-time planar graph algorithm was proposed by Hopcroft and Tarjan [31]. The algorithm is based on Auslander, Parter and Goldstein's algorithm. It starts from a cycle and adding to it one path at a time. Each such new path connects two existing vertices with new edges and vertices. The process continues until either a non-planar subgraph is constructed or the entire graph is constructed. In the former case, the given graph is non-planar; in the latter case, the given graph is planar.

Lempel, Even and Cederbaum [36] used a different approach for planarity testing. Instead of starting with a cycle and adding one path at a time, they start with a single vertex and add one vertex at a time. Each time after a new vertex is added, all the previously added edges that are incident on the new vertex are connected to the vertex; new edges incident on the new vertex are then added with their other endpoints left unconnected. The process continues until a either nonplanar is constructed or the entire graph is completed. Several linear time algorithms based on Lempel, Even and Cederbaum's algorithm had been proposed [6, 17, 46].

2.4.2 Outerplanar Graph

An ***outerplanar graph*** is an undirected graph which can be embedded into the plane so that every vertex lies on the boundary of the exterior face. Obviously, every outerplanar graph is planar, but the converse is not true. K_4 and $K_{2,3}$ (Figures 2.5, 2.7) are the two smallest non-outerplanar graphs. They play a fundamental role in characterizing outerplanar graphs.

Theorem 2. *A graph is outerplanar if and only if it has no subgraph homeomorphic to K_4 or $K_{2,3}$.*

Proof. See [8]. □

Theorem 3. *A graph is outerplanar if and only if each of its biconnected components is outerplanar.*

Proof. See [28]. □

Owing to Theorem 3, many outerpalnar graph algorithms assume that the input graph is biconnected. Brehaut [7], Mitchell [41] and Syslo et al. [47] are such examples. However, if the input graph is not biconnected, a biconnected component algorithm must be used to decompose the input graph into a collection of biconnected components first. This could lengthen the run time of the algorithm significantly. By contrast, both Wiegers [53] and Tsin and Lin [50] do not make such assumption on the input graph.

2.5 Bucket Sort

Bucket sort is a distribution sorting method that is most suitable for sorting d-digit integers or d-tuples of integers in which the integers are bounded by integer k. It runs in linear time providing that k and d are small, fixed constants.

The algorithm works as follows: Let $Array[0..n-1]$ be an array of n d-tuples of integers in which the integers are in the range $\{1, 2, \ldots, k\}$. Then k initially empty buckets are used each of which corresponds to a distinct integer in the given range. The algorithm runs through d iterations. During the $j^{th}, 1 \leq j \leq k$ iteration, a tuple $Array[i] = (a_{i_1}, a_{i_2}, \ldots, a_{i_k})$ is put into bucket $a_{i_{k-j+1}}$. the tuples are then combined into one list with those tuples from bucket i precede those from bucket $i+1$, where $1 \leq i < k$. The list is then used in the following iteration. A brief description of the algorithm is given below.

Algorithm 2 Bucket Sort($Array$, n)

```
for j = 1 to k do
  Bucket[i] := ∅; comment: /* initialize the Buckets */
end for
for j = 1 to d do
  for i = 0 to n − 1 do
    Bucket[a_{i_{k−j+1}}] ← Bucket[a_{i_{k−j+1}}] ⊕ Array[i];
    {comment: Append Array[i] to Bucket a_{i_{k−j+1}}; ⊕ is the concatenation
    operator}
  end for
  Combine the tuples in the buckets into one list such that those tuples from
  bucket i precede those from bucket i + 1, where 1 ≤ i < k;
  Copy the list back into Array[0..n − 1];
end for
```

Chapter 3

A Study of Mitchell's Algorithm

3.1 Maximal Outerplanar Algorithm

Mitchell's algorithm [40] runs in linear time and space. However, it assumes that the given graph is biconnected. If the graph is not biconnected, then a biconnected component algorithm must be used to decompose the graph into a collection of biconnected subgraphs. Michell's algorithm can then be used on each of the subgraphs to find out if any of them is not outerplanar. The given graph is outerplanar if and only if each of its biconnected components is outerplanar. Furthermore, Mitchell's algorithm does not produce an embedding for the given graph if the graph is outerplanar.

Mitchell first presented a linear time and space algorithm for recognizing *maximal* outerplanar graphs. The algorithm is based on the following lemma.

Theorem 4. *A graph $G = (V, E)$ is maximal outerplanar if and only if either G is a triangle or*

i *G contains exactly $2|V| - 3$ edges, and*

ii *G has at least two 2-vertices, and*

iii *no edge of G lies on more than two triangles, and*

iv *for any 2-vertex u, $G - u$ is maximal outerplanar.*

Proof. See [40]. □

The brief description of Mitchell's algorithm is given below:

Given a biconnected undirected graph $G = (V, E)$. LIST is a stack used to store the 2-vertices. EDGES is the set of all the edges in the graph.

1. If $|E| \neq 2|V| - 3$ then stop and report that G is not maximal outerplanar. (Based on Theorem 4(**i**))

2. Push all the vertices of degree 2 onto $LIST$. If the size of $LIST$ is less than 2, then stop and report that G is not maximal outerplanar. (Based on Theorem 4(**ii**))

3. Repeat the following steps until a triangle is left (Based on Theorem 4(**iv**)):

3.1 Pop a 2-vertex $NODE$ from $LIST$;

3.1 Find the vertices $NEAR$ and $NEXT$ which are adjacent to $NODE$;

3.2 Remove $NODE$ from the graph G;

3.3 Add ($NEXT$, $NEAR$) to $PAIRS$;

3.4 If *Deg*($NEXT$) = 2, push $NEXT$ onto $LIST$;
If *Deg*($NEAR$) = 2, push $NEAR$ onto $LIST$;

4. Use two-pass bucket sort to sort $PAIRS$ and $EDGES$ in lexicographical order. (So that Step 5 can be done in $O(|V|)$ time)

5. Compare the lists $PAIRS$ and $EDGES$. If there is an occurrence of an element in $PAIRS$ that is not in $EDGES$, then stop and report that G is not maximal outerplanar. Otherwise, report that the graph G is maximal outerplanar. (Based on Theorem 4(**iii**), there should be one and only one edge between the vertices adjacent to 2-vertices.)

Each time a $2-$vertex is removed from $LISTS$, an edge $(NEXT, NEAR)$ is added to $PAIRS$ indicating that that edge must be an edge in G and hence in set $EDGES$, if G is maximal outerplanar.

3.2 Outerplanar algorithm

Lemma 1. *A graph G is outerplanar if and only if it can be transformed to a maximal outerplanar graph by triangulation* [40].

Owing to Lemma 1, Mitchell's maximal outerplanar algorithm presented in last section can be easily modified to do outerplanar recognition. The complexity of the resulting algorithm is still linear in the number of vertices. The modification involves Steps 1 and 3 only:

Theorem 5. *Let G=(V,E) be an outerplanar graph. Then* $|E| \leq 2|V| - 3$.

Proof. See [28].

□

Owing to Theorem 5, the condition "$|E| \neq 2|V| - 3$" in Step 1 is replaced by "$|E| \not\leq 2|V| - 3$". Step 3 is modified as follows:

3.1 Pop a 2-vertex $NODE$ from $LIST$;

3.2 Find the vertices $NEAR$ and $NEXT$ which are adjacent to $NODE$;

3.3 Remove $NODE$ from the graph G;

3.4 Add ($NEXT$, $NEAR$) to $PAIRS$;

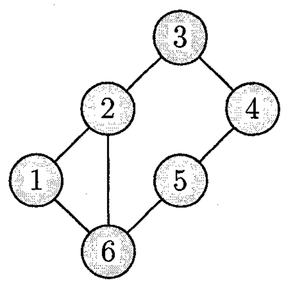
3.5 If edge ($NEXT$, $NEAR$) does not exist in G, add it to $EDGES$ and add $NEAR$ and $NEXT$ to each other's adjacency list;

3.6 If *Deg*($NEXT$) = 2, push $NEXT$ onto $LIST$;
If *Deg*($NEAR$) = 2, push $NEAR$ onto $LIST$;

Step 3.1, 3.2???, 3.4, 3.6 require constant time. Step 3.3???, 3.5 take $O(|V|)$ time.

3.3 An Example of Mitchell's Outerplanar Algorithm

We shall demonstrate an execution of Mitchell's Outerplanar algorithm with the graph depicted in Figure 3.1.



EDGES	LIST	PAIRS
(1,2)	5	ϕ
(1,6)	4	
(2,3)	3	
(3,4)	1	
(4,5)		
(5,6)		
(2,6)		

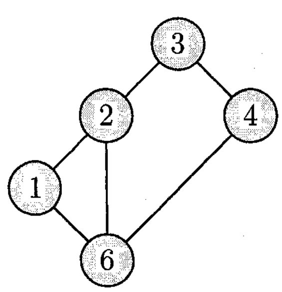
Figure 3.1: An Illustration of Mitchell's Algorithm

3.3.1 Removal of 2-vertices

The algorithm first checks if the condition $|E| \leq 2|V| - 3$ holds. Since the condition holds, all the vertices of degree 2 are pushed onto the stack $LIST$ (see Figure 3.1). As the size of $LIST$ is greater than 2, the algorithm begins to pop the stack $LISTS$.

The first vertex popped out is the vertex 5 (see Figure 3.2). Since vertices 4 and 6 are adjacent to 5, the edge $(4,6)$ is added to $PAIRS$. Since the edge $(4,6)$ does not exist in the graph, it is added to $EDGES$.

$Deg(4)$ and $Deg(6)$ remain unchanged.

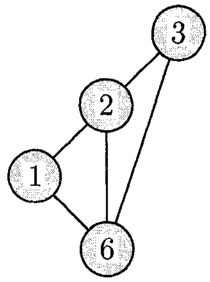


EDGES	LIST	PAIRS
(4,6)	4	(4,6)
(1,2)	3	
(1,6)	1	
(2,3)		
(3,4)		
(4,5)		
(5,6)		
(2,6)		

Figure 3.2: An Illustration of Mitchell's Algorithm: after removal of node 5

The removal of node 4 is similar with node 5. The updated graph, $LIST$, $EDGES$ and $PAIRS$ are shown in Figure 3.3.

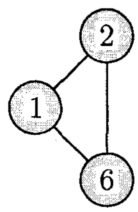
Figure 3.4 shows the graph after node 3 is removed. The difference with pre-



EDGES	LIST	PAIRS
(6,3)	3	(6,3)
(4,6)	1	(4,6)
(1,2)		
(1,6)		
(2,3)		
(3,4)		
(4,5)		
(5,6)		
(2,6)		

Figure 3.3: An Illustration of Mitchell's Algorithm: after removal of node 4

vious step is that (2,6) already exists in the graph, so there is no need to add it into $EDGES$. Since $Deg(2)$ and $Deg(6)$ have changed to 2, there are thus pushed onto the $LIST$.

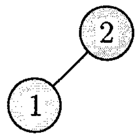


EDGES	LIST	PAIRS
(6,3)	6	(2,6)
(4,6)	2	(6,3)
(1,2)	1	(4,6)
(1,6)		
(2,3)		
(3,4)		
(4,5)		
(5,6)		
(2,6)		

Figure 3.4: An Illustration of Mitchell's Algorithm: after removal of node 3

Figure 3.5 shows the graph after the last removal of vertex from $LIST$ is performed. If the given graph is outerplanar, it would always appear like this: a single edge connect two vertices which are stored at the bottom of $LIST$.

After the process of removing vertices from $LIST$ terminates, the last edge remained in the graph, (2,1), is added to $EDGES$. The current elements of $EDGES$ and $PAIRS$ are shown in Figure 3.6.



EDGES	LIST	PAIRS
(6,3)	2	(1,2)
(4,6)	1	(2,6)
(1,2)		(6,3)
(1,6)		(4,6)
(2,3)		
(3,4)		
(4,5)		
(5,6)		
(2,6)		

Figure 3.5: An Illustration of Mitchell's Algorithm: after removal of node 6

EDGES	PAIRS
(2,1)	(1,2)
(6,3)	(2,6)
(4,6)	(6,3)
(1,2)	(4,6)
(1,6)	
(2,3)	
(3,4)	
(4,5)	
(5,6)	
(2,6)	

Figure 3.6: An Illustration of Mitchell's Algorithm: *PAIRS* and *EDGES* after all the 2−vertices are removed

3.3.2 Bucket Sort

Both the lists *EDGES* and *PAIRS* can be sorted by a two-pass Bucket Sort. Each pair in *EDGES* and *PAIRS* consists of two integers from 1 to 6. Before sorting, every pair is adjusted so that the first integer is no greater than the second integer.

The arrays in Figure 3.7 are then sorted using 2-pass Bucket sort. The buckets are labeled from 1 to 6. In the first pass, each pair in *PAIRS* (*EDGES*, respectively) is put into a bucket whose label is identical to the second integer of the ordered pair. A partially sorted *PAIRS* (*EDGES*, respectively)(sorted by their second integer) is obtained. In the second pass, each pair in *PAIRS* (*EDGES*, respectively) is put into a bucket whose label is identical to the first integer of the ordered pair. A sorted *PAIRS* (*EDGES*, respectively) is then obtained. Fig-

EDGES	PAIRS
(1,2)	(1,2)
(1,2)	(2,6)
(3,6)	(3,6)
(4,6)	(4,6)
(1,2)	
(1,6)	
(2,3)	
(3,4)	
(4,5)	
(5,6)	
(2,6)	

Figure 3.7: An Illustration of Mitchell's Algorithm: *PAIRS* and *EDGES* before Bucket Sort

ures 3.8 and 3.9 show the results of Bucket sort.

EDGES	PAIRS
(1,2)	(1,2)
(1,2)	(2,6)
(1,2)	(3,6)
(2,3)	(4,6)
(3,4)	
(4,5)	
(3,6)	
(4,6)	
(1,6)	
(5,6)	
(2,6)	

Figure 3.8: An Illustration of Mitchell's Algorithm: *PAIRS* and *EDGES* after one-pass Bucket Sort

3.3.3 Check PAIRS and EDGES

After both *PAIRS* and *EDGES* are sorted, the two lists are scanned to determine if every pair in *PAIRS* also appears in *EDGES*. For the given example, all the pairs $(1,2), (2,6), (3,6), (4,6)$ are in *EDGES*. The give graph is thus outerplanar.

EDGES	PAIRS
(1,2)	(1,2)
(1,2)	(2,6)
(1,2)	(3,6)
(1,6)	(4,6)
(2,3)	
(2,6)	
(3,4)	
(3,6)	
(4,5)	
(4,6)	
(5,6)	

Figure 3.9: An Illustration of Mitchell's Algorithm: *PAIRS* and *EDGES* after a two-pass Bucket Sort

3.4 Implementation

Unfortunately, the presentation of Mitchell's outerplanar algorithm in Mitchell's original paper [41] is very brief. It is not at all clear that the algorithm can be implemented in linear time and space. For instance, in Step 3.5, the algorithm has to check whether the edge $(NEAR, NEXT)$ already exists in the graph G before it is added to $EDGES$. This could be accomplished by scanning the adjacency list of $NEXT$ for the vertex $NEAR$. The vertex $NEAR$ appears in the adjacency list if and only if the edge $(NEAR, NEXT)$ exists in G. Since it takes $O(|V|)$ time to search an adjacency list in the worst case, if there are $O(|V|)$ $NEXT$s, the algorithm would take $O(|V|^2)$ time rather than linear time.

3.4.1 Our strategies in the implementation

We adopt the following strategies in implementing Mitchell's algorithm:

- The adjacency list data structure is used to represent the input graph $G = (V, E)$.
- Delay checking if the edge $(NEXT, NEAR)$ exists in the given graph until either $NEXT$ or $NEAR$ is popped out of $LIST$ (i.e. $Deg(NEXT) = 2$ or $Deg(NEAR) = 2$).
- In order to save the time on scanning the adjacency list, we shorten the adjacency list by deleting all the nodes of degree 0 . As we often deal with

node with degree 2, it takes only $O(1)$ to scan this adjacency list.

3.4.2 Main Steps of our Implementation

We briefly describe the main steps of our implementation first.

1. Check whether $|E| \leq 2|V| - 3$. If not, then Stop.
2. Push all the vertices with degree 2 onto $LIST$. If $size(LIST) < 2$, then Stop.
3. Pop $NODE$ from $LIST$; Find the vertices $NEAR$ and $NEXT$ which are adjacent to $NODE$; Add $(NEXT, NEAR)$ to $PAIRS$; Remove $NODE$ from the graph.
4. Add $NEAR$ and $NEXT$, each with a *mark*, to each other's adjacency list.
5. If $Deg(NEXT) = 2$, check if any node with a mark in the adjacency list is a duplicate entry, if it is, then delete the node with a mark; otherwise add the node to adjacency list of $NEXT$ and update $Deg(NEXT)$ accordingly. Do the same for vertex $NEAR$ if $Deg(NEAR) = 2$.
6. If $Deg(NEXT) = 2$ or $Deg(NEAR) = 2$, push it onto $LIST$.
7. Use a two-pass Bucket Sort on PAIRS and EDGES.
8. If there is an occurrence of an element in $PAIRS$ that is not in $EDGES$, then Stop, else report that the given graph G is outerplanar.

The changes take place in Step 4 and 5. To save the efficiency, our algorithm does not check whether the edge (NEAR,NEXT) exists in adjacency list until $Deg(NEXT)$ or $Deg(NEAR) = 2$. In this way, it takes only $O(2)$ times in stead of $O(|V|)$ in Mitchell's algorithm.

In order to record NEXT or NEAR which may be added to adjacency list later, we add this node with a mark (denoted by $node^*$) at the beginning of the adjacency list, which is faster than at the end. Then, we do a "CheckExist" process when Deg(NODE) = 2. The steps contained in the "CheckExist" process are: If we find out that the edge has already existed before the node* is added, the algorithm would delete the node*. Otherwise, the node* would be deleted from the beginning of the adjacency list and a regular node would be added to the end of adjacency list.

3.4.3 A Detailed Implementation

The input graph is represented by the adjacency lists of its vertices. Each node in the adjacency list of a vertex v contains a vertex that is adjacent to v and hence also represents an edge incident on v. To distinguish between a marked node and a regular node, we color the nodes with different colors. Specifically, if the node is colored white, then it is a regular node; if it is colored red, then it a marked node. Marked node are inserted at the beginning of the adjacency list. Details are spelled out in Algorithms 3, 5, 6 and 7 below.

Algorithm 3 An Implementation of Mitchell's Outerplanar Algorithm

if $(|E| \leq 2|V| - 3)$ **then**
  Output "No"
end if;
$LIST \leftarrow \{v|Deg[v] = 2\}$; $PAIRS \leftarrow \emptyset$;
if $(|LIST| < 2)$ **then**
  Output "No"
end if;
for $L = 1$ **to** $|V| - 2$ **do**
  $NODE \leftarrow$ **pop**$(LIST)$;
  $NEAR, NEXT \leftarrow$ the two vertices adjacent to $NODE$;
  Add $(NEAR, NEXT)$ to list $PAIRS$;
  Remove $NODE$ from the graph;
  Decrement $Deg(NEAR)$ and $Deg(NEXT)$;
  if $(Deg(NEAR) \leq 2)$ **then**
    $ChkAdj(NEAR, NEXT)$;
  end if
  if $(Deg(NEXT) \leq 2)$ **then**
    $ChkAdj(NEXT, NEAR)$;
  end if;
  if $(Deg(NEAR) > 2) \wedge (Deg(NEXT) > 2)$ **then**
    $AddRed(NEXT, NEAR)$;
  end if
  if $(Deg(NEAR) \leq 2)$ **then** Add $NEAR$ to $LIST$;
  if $(Deg(NEXT) \leq 2)$ **then** Add $NEXT$ to $LIST$;
  if $(|LIST| - L < 2)$ **then**
    Output "No"
  end if
end for;
Add the edge $(NEAR, NEXT)$ to $EDGES$;
Lexicographically sort $EDGES$;
Lexicographically sort $PAIRS$;
if there is an edge in $PAIRS$ and not in $EDGES$ **then**
  Output "No"
else
  Output "Yes"
end if

Algorithm 4 Check the adjacency list of vertex a for vertex b

Procedure *ChkAdj(a,b)*
  if (there is no b colored white in the adjacency list of a) **then**
    $AddWhite(a, b)$;
  end if;
  for (each vertex v in the adjacency list of a) **do**
    if ($Deg[v] = 0$) **then** Remove v from the list;
    else if (v is red) **then**
        if ($\nexists$ another v colored white in the list) **then**
          $RemoveRed((a, v))$;
          $AddWhite(a, v)$;
        else $RemoveRed(a, v)$;

Algorithm 5 Add White Node

Procedure *AddWhite(a,b)*
  Add the edge (a,b) to list $EDGES$
  Add a and b with color white to the end of each other's adjacency list
  Increment($Deg(a)$); Increment($Deg(b)$)

In Step 1, if $|E| > 2|V| - 3$, then by Theorem 5, the input graph cannot be outerplanar. The algorithm thus terminates its execution and outputs a "No".

In Step 4, the set of vertices of degree 2 are pushed onto the stack $LIST$. The list of edges $PAIRS$ is initialized to the empty set.

In Step 9, a vertex $NODE$ is popped out of the stack $LIST$. Since $NODE$ is of degree 2, it can have only two adjacent vertices, $NEAR$ and $NEXT$.

In Step 10, the edge $(NEXT, NEAR)$ is added to $PAIRS$.

In Step 11, vertex $NODE$ is removed from the graph by setting $Deg(NODE)$ to 0.

In Step 12, the degrees of $Deg(NEXT)$ and $Deg(NEAR)$ are incremented according.

In Steps $13-15$, if the $Deg(NEAR) \leq 2$, then its adjacency list is scanned for $NEXT$. The existence of a $NEXT$ vertex colored white indicates that the edge $(NEAR, NEXT)$ exists in G. So, no further action is necessary. Otherwise, a vertex $NEXT$ ($NEAR$, respectively) colored white is added to the adjacency list

Algorithm 6 Add Red Node

Procedure *AddRed(a,b)*

Add b with color red to the beginning of a's adjacency list

Algorithm 7 Remove Red Node

Procedure *RemoveRed(a,b)*

Remove b (colored red) from the adjacency list of a

of $NEAR$ ($NEXt$, respectively). This effectively adds the edge $(NEAR, NEXT)$ to G. Therefore, the edge $(NEAR, NEXT)$ is also added to $EDGES$ and the degrees of a and b are incremented accordingly. Next, the adjacency list of $NEAR$ is scanned. For each vertex v in the list, if $Deg(v) = 0$, vertex v is removed rom the list. If v is colored red and there is a vertex v colored white in the list, then the red v is removed; otherwise, the red v is removed, and a white v is added to the adjacency of $NEAR$ while a white $NEAR$ is added to the adjacency list of v. Moreover, the edge $(NEAR, v)$ is added to $EDGES$ and $Deg(v)$ and $Deg(NEAR)$ are incremented accordingly.

Steps 16-18 are similar to Steps 13-15.

Steps $19-21$, if neither $Deg(NEAR) \leq 2$ nor $Deg(NEXT) \leq 2$, then a vertex $NEXT$ ($NEAR$, respectively) colored red is added to the adjacency list of $NEAR$ ($NEXT$, respectively). When $Deg(NEAR)$ ($Deg(NEXT)$, respectively) finally becomes two or less, the red $NEXT$ ($NEAR$, respectively) will be processed in Steps 13-15 (16-19, respectively).

In Steps 22 and 23, vertex $NEAR$ ($NEXT$, respectively) is pushed onto the stack $LIST$ if $Deg(NEAR) \leq 2$, ($Deg(NEXT) \leq 2$, respectively)

In Steps $24-26$, If there are less then two vertices on the stack $LIST$ and fewer than $|V|-2$ vertices had been popped out of $LITS$, then execution of the algorithm terminates and the graph G is reported as non-outerplanar.

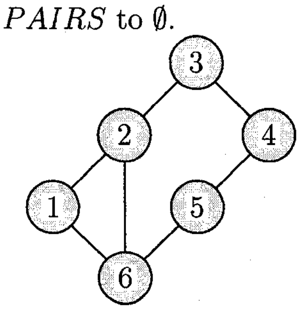
In Steps 29-35, both $EDGES$ and $PAIRS$ are sorted lexicographically using bucket sort. This is to ensured that checking if $PAIRS \subseteq EDGES$ can be carried out in linear time. The details are given in Algorithm 8 below.

Algorithm 8 Check if PAIRS $\subseteq$ EDGES

```
{Let PAIRS = {pair_i | 1 <= i <= n}, EDGES = {edge_j | 1 <= j <= m}}
j = -1;
for i := 0 to n do
  j++;
  if j > m then
    Output "NO"; stop
  end if;
  while pair_i != edge_j do
    j++;
    if j > m then
      Output "NO"; stop
    end if
  end while
end for
Output "YES".
```

3.4.4 An Illustration of Mitchell's Outerplanar Algorithm

We use the example in Figure 3.10 to illustrate Mitchell's algorithm for outerplanarity testing. After reading the input graph file, the Adjacency Lists and the elements in $EDGES$ would be as shown in Figure 3.10. The algorithm starts with verifying $|E| \leq 2|V| - 3$. Since $|V| = 6$ and $|E| = 7$, the condition is satisfied. The next step is to push all vertices that are of degree 2 onto $LIST$ and initialize $PAIRS$ to $\emptyset$.



EDGES
(1,2)
(1,6)
(2,3)
(3,4)
(4,5)
(5,6)
(2,6)

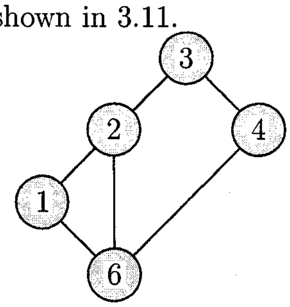
Adjacency List
1 : 2 6
2 : 1 6 3
3 : 2 4
4 : 3 5
5 : 4 6
6 : 1 2 5

LIST
5
4
3
1

PAIRS
ϕ

Figure 3.10: An Illustration of Mitchell's Outerplaner Algorithm; $|V| = 6$.

Node 5 is the first vertex popped out of $LIST$ and removed from G. Since the two vertices adjacent to vertex 5 are vertices 4 and 6. the edge (4,6) is added to $LISTS$. Moreover, as $Deg(4) < 2$ after vertex 5 is removed, the adjacency list of vertex 4 is examined. As the list does not contain an unmarked vertex 6 (i.e. a white vertex 6), vertex 6 is thus added at the end of the adjacency list of vertex 4 while vertex 4 is added at the end of the adjacency list of vertex 6. Furthermore, vertex 5 in the adjacency list of vertex 4 is removed. The updated information is shown in 3.11.



EDGES

(4,6)
(1,2)
(1,6)
(2,3)
(3,4)
(4,5)
(5,6)
(2,6)

Adjacency List

1 : 2 6
2 : 1 6 3
3 : 2 4
4 : 3 6
6 : 1 2 5 4

LIST

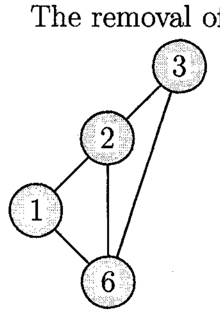
4
3
1

PAIRS

(4,6)

Figure 3.11: An Illustration of Mitchell's Algorithm: After removal of vertex 5

The removal of vertex 4 is similar to vertex 5 (see Figure 3.12).



EDGES

(3,6)
(4,6)
(1,2)
(1,6)
(2,3)
(3,4)
(4,5)
(5,6)
(2,6)

Adjacency List

1 : 2 6
2 : 1 6 3
3 : 2 6
6 : 1 2 5 4 3

LIST

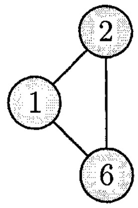
3
1

PAIRS

(3,6)
(4,6)

Figure 3.12: An Illustration of Mitchell's Algorithm: After removal of vertex 4

The next vertex popped out of $LISTS$ is vertex 3. The edge (2,6) is then added to $PAIRS$. After vertex 3 is deleted, both $Deg(2)$ and $Deg(6)$ become 2. Suppose vertex 2 is examined first, then the adjacency list of vertex 2 is scanned and vertex 3 is removed. Furthermore, as an unmarked vertex 6 appears in the list, no edge $(2,6)$ is added to $EDGES$. Vertex 6 is then examined and its adjacency list is scanned. Since $Deg(3) = Deg(4) = Deg(5) = 0$, all these vertices are removed. Since an unmarked vertex 2 appears in the list, no edge $(2,6)$ is added to $EDGES$. Vertices 2 and 6 are the pushed onto $LIST$ (see Figure 3.13).

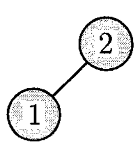


EDGES	Adjacency List	LIST
(3,6)		
(4,6)		
(1,2)		
(1,6)	1 : 2 6	6
(2,3)	2 : 1 6	2
(3,4)	6 : 1 2	1
(4,5)		**PAIRS**
(5,6)		
(2,6)		(2,6)
		(3,6)
		(4,6)

Figure 3.13: An Illustration of Mitchell's Algorithm: After removal of vertex 3

The next vertex popped out of $LIST$ is 6. The edge $(1,2)$ is then added to $PAIRS$. Since $Deg(1) < 2$, the adjacency list of vertex 1 is scanned and vertex 6 is removed from the list. Furthermore, as there is an unmarked vertex 2 in the list, no edge $(1,2)$ is added to $EDGES$. Similarly, as $Deg(2) < 2$, the adjacency list of vertex 2 is scanned and vertex 6 is removed from the list. Finally an edge $(1,2)$ is added to $EDGES$.

Finally, as $PAIRS \subseteq EDGES$, the algorithm thus terminates execution with a "Yes".



EDGES	Adjacency List	LIST
(1,2)		
(3,6)		
(4,6)		
(1,2)		
(1,6)	1 : 2	2
(2,3)	2 : 1	1
(3,4)		**PAIRS**
(4,5)		
(5,6)		(1,2)
(2,6)		(2,6)
		(3,6)
		(4,6)

Figure 3.14: An Illustration of Mitchell's Algorithm: After removal of vertex 6

Chapter 4

A Study of Wiegers' Algorithm

4.1 Outerplanar algorithm

In contrast with Mitchell's algorithm, Wiegers' Outerplanar algorithm [53] accepts non-biconnected graphs as the input graph and performs no sorting. This algorithm uses a $2-$reducible graph testing and an edge-coloring technique. Similar to Mitchell's algorithm, Wiegers' algorithm repeatedly removes vertices of degree two or less from the graph; whenever a vertex of degree two is removed, a new edge joining its two neighbors is added to the graph if the edge does not exist. If the algorithm runs out of vertices of degree two or less before reducing the input graph into an edgeless graph, the algorithm terminates its execution and reports that the graph is non-outerplanar. This is because the graph must contain a subgraph that is homeomorphic to K_4. The edge-coloring technique is used to keep track of the number of triangles each edge belongs to. If any edge belongs to more than two triangles, the algorithm would report that the graph is non-outerplanar indicating that the graph contains a subgraph that is homeomorphic to $K_{2,3}$.

4.1.1 The 2-Reducible Graph Algorithm

Definition 33. [53] *A graph G=(V,E) is* $2-$***reducible*** *if and only if*

$E = \emptyset$, *or*

$\exists u \in V$ *such that* $Deg(u) \leq 1$, $G_u = G - \{u\}$ *is 2-reducible, or*

$\exists u \in V$ *such that* $Deg(u) = 2$ *and* v_1, v_2 *are the adjacent vertices of* v, $G_u = (V - \{u\}, E - \{(u, v_1), (u, v_2)\} \cup \{(v_1, v_2)\})$ *is 2-reducible.*

Theorem 6. *The class of outerplanar graphs $\subseteq$ the class of 2-reducible graphs $\subseteq$ planar graphs.*

Proof. [53]. $\square$

A 2−reducible graph can be totally disconnected or can be made totally disconnected by repeatedly deleting edges adjacent to vertices of degree at most 2. Wiegers showed that a 2-reducible graph can be recognized in $O(|V|)$ time. Based on Theorem 6, an outerplanar graph is a 2-reducible graph, but the converse is not true.

Since it is both annoying and time consuming to check whether there exists an edge between two given vertices u and v, Wiegers' 2-Reducible graph algorithm would not do such checking until the degree of one of the two vertices becomes less than 3. Therefore, two adjacent lists $AList'(u)$ and $ALIst'(v)$ are maintained to hold this potential edge. When the degree of u or v becomes less than 3, the edge (u, v) is then moved from $AList'(u)$ ($ALIst'(v)$, respectively) to $AList(u)$ ($ALIst(v)$, respectively).

The following is a brief description of the 2-Reducible graph algorithm:

1. Given a graph $G = (V, E)$, check whether $|E| > 2|V| - 3$. If yes, then the graph is not outerplanar;

2. Let M be the set containing all the vertices of degree less then or equal to 2 during the execution;

3. Remove one vertex $u \in M$. If $AList'(u)$ contains a vertex v, then remove v from $AList'(u)$. Furthermore, if v does not appears in $AList(u)$, then v (u, respectively) are inserted into $AList(u)$ ($AList(v)$, respectively) which effectively adds the edge (u, v) to the graph. Vertex u is returned to M if $Deg(u) \leq 2$. On the other hand, if $AList'(u)$ is empty, vertex u would be made an isolated vertex. Moreover, if $Deg(u) = 2$, and v and w are the two adjacent vertices of u, then vertex v (w, respectively) is inserted into $AList'(w)$ ($AList'(v)$, respectively). Finally, if $Deg(v) \leq 2$, v is added to M. The same applies to vertex w.

4. When $M = \emptyset$, $|E| = 0$ if and only if G is outerplanar.

The 2-Reducible graph algorithm is presented as Algorithm 9.

Algorithm 9 2-Reducible Graph Algorithm

```
if |E| > 2|V| - 3 then
  return false
end if
M ← {u | Deg(u) ≤ 2};
while M ≠ ∅ do
  Remove u from M;
  if Deg(u) ≤ 2 then
    if AList'(u) ≠ ∅ then
      Remove u_1 from AList'(u);
      if u_1 ∉ AList(u) then
        add u_1 to AList(u); add u to AList(u_1)
        Increment Deg(u_1); Increment Deg(u);
      end if
      if (Deg(u) ≤ 2) then M ← M ∪ {u};
    else
      if Deg(u) = 1 then
        Let u_1 ∈ AList(u), delete u_1 from AList(u);
        Decrement Deg(u_1);
        if (Deg(u_1) ≤ 2) then M ← M ∪ {u_1};
      else
        if Deg(u) = 2 then
          Let u_1, u_2 ∈ AList(u); Remove u_1, u_2 from AList(u);
          Add u_1 in AList'(u_2); Add u_2 in AList'(u_1);
          Decrement Deg(u_1); Decrement Deg(u_2);
          if (Deg(u_1) ≤ 2) then M ← M ∪ {u_1};
          if (Deg(u_2) ≤ 2) then M ← M ∪ {u_2};
        end if
      end if
    end if
  end if
end while
return |E| = 0
```

4.1.2 The Edge Coloring Technique

Wiegers classified the edges in an outerplanar graph into three types: *cross edge*, *outer edge* and *bridge*. Each edge in the outerplanar graph can belong to at most two triangles. Note that if an edge belongs to a triangle, then the other two edges of the triangle corresponds to a non-trivial path connecting the endpoints of that edge in G. Therefore, if an edge belongs to three triangle, then there exist three edge-disjoint paths in G connecting the endpoints of that edge. The three paths form a subgraph of G which is homeomorphic to $K_{2,3}$. The graph G is thus non-outerplanar. The edge-coloring technique is used to keep track of the number of times each edge appears on a triangle in the course of executing the algorithm. Specifically, a cross edges is an edge for which no triangle containing it has been discovered. An outer edge is an edge for which one triangle containing it has been discovered. A bridge is an edge for which either no triangle or two triangles containing it have been discovered. In the former case, it is a genuine bridge (i.e a cut-edge), In the latter case, it implies that if a triangle containing the edge is discovered at a later stage, then the graph G contains a subgraph that is homeomorphic to $K_{2,3}$. The graph is thus non-outerplanar and the coloring is called an *unacceptable edge coloring.* The outerplanar graph algorithm is a modification of the 2-reducible graph algorithm using the edge-coloring technique. It is based on the following idea: every reduction of an outerplanar graph with an acceptable edge coloring gives rise to an outerplanar graph with an acceptable edge coloring. If an unacceptable edge coloring is created by such reduction, the graph is non-outerplanar.

Definition 34. $\forall (a,b) \in E$, ***col(a,b)*** *denotes the color assigned to the edge* (a,b), *which can be cross, outer or bridge.*

Remark. $\forall (a,b) \in E, col(a,b)$, $col(a,b)$ *is initialized to* cross. *The value of* $col(a,b)$ *is updated whenever a reduction is applied to a vertex during the execution of the outerplanar graph algorithm.*

In the following discussion, u is the vertex removed from M. If $Deg(u) = 1$, then u_1 is the vertex adjacent to u. If $Deg(u) = 2$, then u_1 and u_2 are the two vertices adjacent to u. Finally, $A = \{cross, outer, bridge\}$ and $B = \{cross, outer\}$.

We shall explain how to use the edge-coloring technique in conjunction with vertex reduction to determine if a graph $G = (V, E)$ is outerplanar. Seven cases are to be considered separately and are summarized in Table 4.1.

In Case (i), $Deg(u) = 1$. In Cases (ii) and (iii), $Deg(u) = 2$ and $(u_1, u_2) \notin E$. In Cases (iv) to (vii), $Deg(u) = 2$ and $(u_1, u_2) \in E$. In the first case, the edge (u, u_1) is simply discarded; no coloring of edges is necessary. In the remaining six cases, the color of the edge (u_1, u_2) must be determined.

Deg(u)=1	col$(u, u_1) \in A$	acceptable (Figure 4.1)	
Deg(u)=2	$(u_1, u_2) \notin E$	col$(u, u_1) \in B$, col$(u, u_2) \in B$	acceptable (Figure 4.2)
		col$(u, u_1) \in A$, col(u, u_2)=bridge	acceptable (Figure 4.3)
	$(u_1, u_2) \in E$	col$(u, u_1) \in B$, col$(u, u_2) \in B$, (u_1, u_2)=cross	acceptable (Figure 4.4
		col$(u, u_1) \in B$, col$(u, u_2) \in B$, (u_1, u_2)=outer	acceptable (Figure 4.5)
		col$(u, u_1) \in B$, col$(u, u_2) \in B$, (u_1, u_2)=bridge	unacceptable (Figure 4.6)
		col$(u, u_1) \in A$, col(u, u_2)=bridge, $(u_1, u_2) \in A$	unacceptable (Figure 4.7)

Table 4.1: Types of reduction

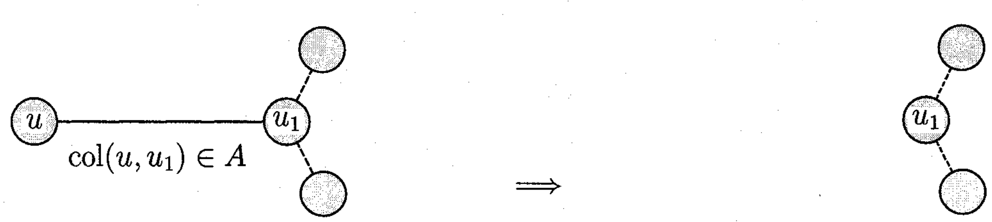


Figure 4.1: case (i):$Deg(u) = 1$. No matter col(u, u_1) is cross, outer or bridge, G remains having acceptable coloring

In Case (ii), $col(u, u_1), col(u, u_2) \in \{cross, outer\}$ and $(u_1, u_2) \notin E$. Since $col(u, u_1), col(u, u_2) \in \{cross, outer\}$, therefore the edge (u, u_1) ((u, u_2), respectively) lies on at most one triangle. It follows that the new edge (u_1, u_2) lies on at most one triangle. It is thus assigned the color *outer*. The coloring for the graph after the vertex u is removed and the edge (u_1, u_2) is added is thus an acceptable coloring (see Figure 4.2).

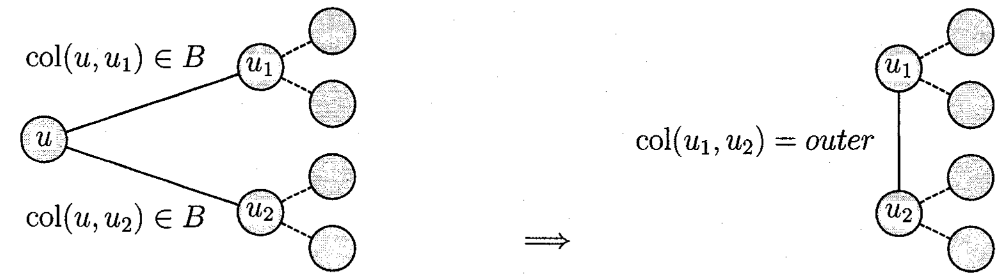


Figure 4.2: case (ii):$Deg(u) = 2$, u_1 and u_2 are not joined with an edge.

In Case (iii), $col(u, u_2) \in \{bridge\}$ while $col(u, u_1)$ can be any of the three colors and $(u_1, u_2) \notin E$. Then $col(u, u_2) \in \{bridge\}$ implies that the edge (u, u_2) lies on two triangles or no triangle while $col(u, u_1) \in \{cross, outer, bridge\}$ implies that the edge (u, u_1) lies on at most two triangles. As a result, if the new edge (u_1, u_2) is added in, the edge cannot lie on any triangle in the graph after a reduction is applied to vertex u. It is thus assigned the color *bridge*. Note that if both (u, u_1) and (u, u_2) are genuine bridges, then the new edge would also be a genuine bridge in the graph after a reduction is applies to vertex u. The coloring for the graph after the reduction is thus an acceptable coloring (see Figure 4.3).

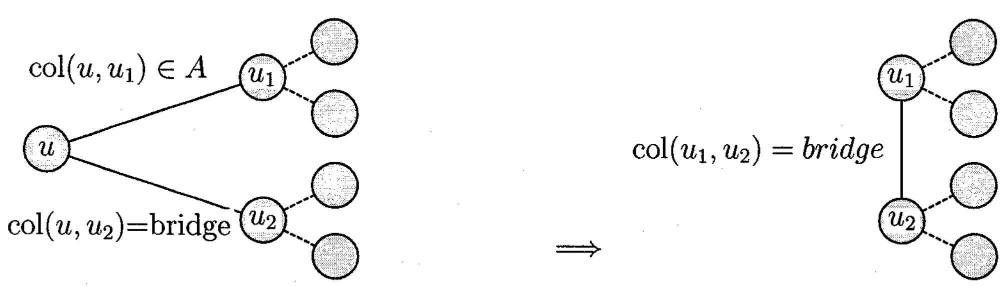


Figure 4.3: case (iii):$Deg(u) = 2$, u_1 and u_2 are not joined with an edge.

In Case (iv), $col(u, u_1), col(u, u_2) \in \{cross, outer\}$ and $(u_1, u_2) \in \{cross\}$. Since $col(u, u_1), col(u, u_2) \in \{cross, outer\}$, the edge (u, u_1) ((u, u_2), respectively) lies on at most one triangle. $(u_1, u_2) \in \{cross\}$ implies that it lies on no triangle so far. It follows that the edge (u_1, u_2) lies on at most one triangle in the graph after a reduction is applied to u. It is thus assigned the color *outer*. The coloring for the graph after the reduction is thus an acceptable coloring (see Figure 4.4).

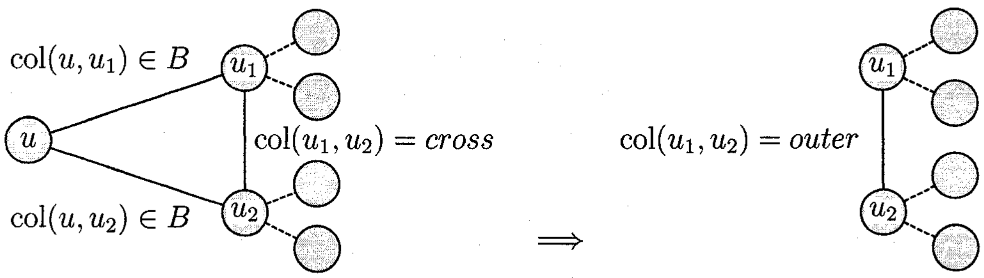


Figure 4.4: case (iv):$Deg(u) = 2$, u_1 and u_2 are joined with an edge.

In Case (v), $col(u, u_1), col(u, u_2) \in \{cross, outer\}$ and $(u_1, u_2) \in \{outer\}$. Since $col(u, u_1), col(u, u_2) \in \{cross, outer\}$, the edge (u, u_1) ((u, u_2), respectively) lies on at most one triangle. $(u_1, u_2) \in \{outer\}$ implies that it lies on one triangle. It follows that the edge (u_1, u_2) lies on two triangles in the graph after a reduction is applied to u. It is thus assigned the color *bridge*. The coloring for the graph after the reduction is thus an acceptable coloring (see Figure 4.5).

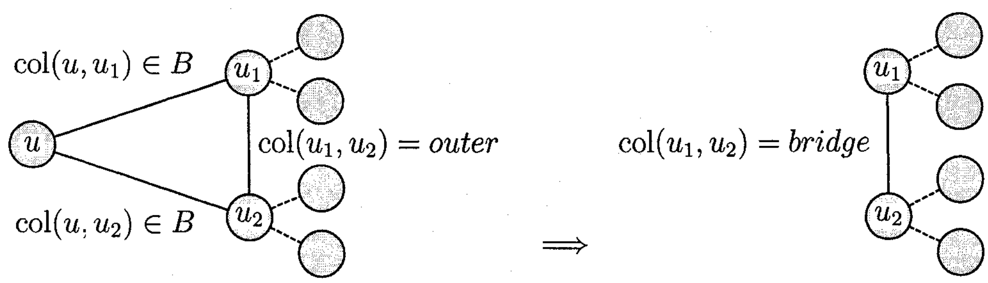


Figure 4.5: case (iv):$Deg(u) = 2$, u_1 and u_2 are joined with an edge.

In Cases (vi) and (vii), at least one of the three edges $(u, u_1), (u, u_2)$ and (u_1, u_2) is colored *bridge*. This implies that the edge lies on two triangles so far. Since the three edges form a third triangle containing the edge, the edge thus lies on three triangles. It follows that the is a subgraph of G that is homeomorphic to $K_{2,3}$. The coloring for the graph is thus an unacceptable coloring (see Figure 4.6 and 4.7).

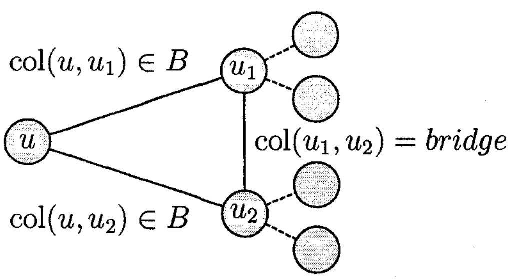


Figure 4.6: case (v):$Deg(u) = 2$, u_1 and u_2 are joined with an edge.

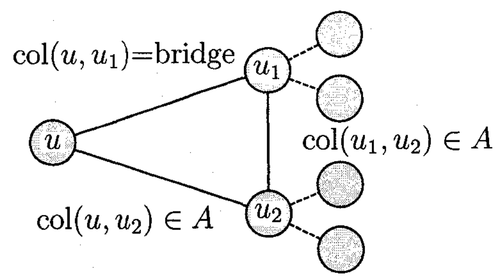


Figure 4.7: case (vi):$Deg(u) = 2$, u_1 and u_2 are joined with an edge.

4.2 Implementation

The doubly-linked adjacency list is required to represent the graph. Furthermore, cross-pointers are used between adjacency lists in order to saves time when an edge is to be deleted from the graph. The deletion of an edge (u, v) (assuming $Deg(u) \leq 2$) consists two steps: first, find v in $AList(u)$ and remove it. As $Deg(u) \leq 2$, this step takes $O(1)$ time. Next, use the cross pointer to locate u in $AList(v)$ and remove it. This clearly takes O(1) time.

Let $A = \{cross, outer, bridge\}$, $B = \{cross, outer\}$, $col(a, b)$ is the color of the edge $(a,\ b)$ in *AList*, and $col'(a, b)$ is the color of $(a,\ b)$in $AList'$. Our implementation is shown in Algorithm 10.

In Step 1, after loading the input graph file, we check if $|E| \leq 2|V| - 3$ is satisfied.

In Step 4, each edge in the *AList* is associated with a color. At the beginning, the color is initialized to *cross*.

Algorithm 10 Implementation of Wiegers' Outerplanar Graph Algorithm

```
if |E| > 2|V| − 3 then
   return false
end if
for every edge (a, b) ∈ E do
   col(a, b)= cross
end for
M ← {u ∈ V|Deg(u) ≤ 2};
while M ≠ ∅ do
   Remove u from M
   if Deg(u) ≤ 2 then
      if AList'(u) ≠ ∅ then
         MoveEdge(u)
      else
         if Deg(u)=1 then
            Let u1 ∈ AList(u), remove u1 from AList(u);
            Decrement Deg(u1);
            if (Deg(u1) ≤ 2) then M ← M ∪ {u1};
         else
            if Deg(u) = 2 then
               Let u1, u2 ∈ AList(u), delete u1, u2 from AList(u);
               Decrement Deg(u1); Decrement Deg(u2);
               if (Deg(u1) ≤ 2) then M ← M ∪ {u1};
               if (Deg(u2) ≤ 2) then M ← M ∪ {u2};
               Add u1 in AList'(u2); Add u2 in AList'(u1);
               if col(u, u1), col(u, u2) ∈ B then
                  col'(u1, u2) ← outer
               else
                  col'(u1, u2) ← bridge
               end if
            end if
         end if
      end if
   end if
end while
return |E| = 0
```

Algorithm 11 MoveEdge

```
Procedure MoveEdge(u)
 1. Let u1 ∈ AList'(u), delete u1 from AList'(u)
 2. if u1 ∈ AList(u) then
 3.   if col'(u, u1) = bridge then
 4.     return false
 5.   else
 6.     if col(u, u1) = cross then
 7.       col(u, u1) ← outer
 8.     else
 9.       if col(u, u1) = outer then
10.         col(u, u1) ← bridge
11.       else
12.         if col(u, u1) = bridge then
13.           return false
14.         end if
15.       end if
16.     end if
17.   end if
18. else
19.   Insert u into AList(u1); Insert u1 into AList(u); col(u, u1) ← col'(u, u1);
20. end if
21. M = M ∪ {u} if Deg(u) ≤ 2;
```

In Step 7, let M be the set containing all the vertices with degree 2 or less.

In Step 8, the **while** loop will iterate until M is empty.

In Steps 9 to 11, we select one vertex u from M. If $AList'(u)$ is not empty, then Procedure *MoveEdge*(u) is invoked.

In Steps 14 to 17, when $Deg(u) = 1$, $Deg(u)$ is reduced to 0 and u_1 is removed from $AList(u)$ which takes $O(1)$ time. Using the cross-pointer in the adjacency lists, we can locate the vertex u in $AList(u_1)$ and remove u from $AList(u_1)$ in O(1) time.

In Steps 19 to 28, as $Deg(u) = 2$, we immediately find the two vertices in $AList(u)$ and the colors associated with them. If both col(u, u_1) and col(u, u_2) are *cross* or *outer*, then as shown in cases (ii), (iv), (v), (vi), we add (u_1, u_2) to $AList'$ and color it as *outer*. Otherwise, by case (iii), the edge (u_1, u_2) is added to $AList'$ and colored as *bridge*. $Deg(u)$ is reduced to 0 and u_1, u_2 are both deleted from $AList(u)$. Finally, vertex u is deleted from both $AList(u_1)$ and $AList(u_2)$.

In Procedure *MoveEdge*:

- In Step 1, a vertex u_1 is removed from $AList'(u)$. If $(u, u_1) \notin E$, then u_1 is added to $AList(u)$ and $col'(u, u_1)$ is assigned to $col(u, u_1)$.

- In Steps 3 and 4, Case (vii) occurs which implies that the edge coloring is unacceptable. The graph is thus non-outerplanar.

- In Steps 6 and 7, Case (iv) occurs which implies that the edge coloring is acceptable.

- In Steps 9 and 10, Case (v) occurs which implies that the edge coloring is acceptable.

- In Steps 12 and 13, Case (vi) occurs which implies that the edge coloring is unacceptable. The graph is thus non-outerplanar.

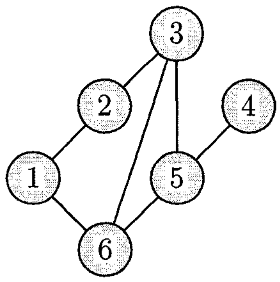
4.2.1 An Example

We present an example of the implementation of Wiegers' outerplanar graph algorithm. We shall use crs, out, brg as the abbreviations of cross edge, outer edge and bridge, respectively.

As shown in Figure 4.8, we display the contents of M and $AList$. Initially, the colors of all the edges are initialized to cross edge and M consists of the vertices with degree 2 or less.

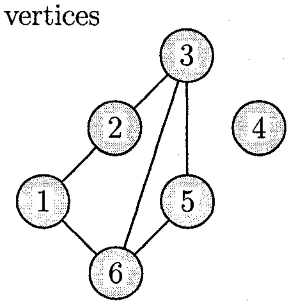
In Figure 4.9, vertex 4 is selected from M. Since $Deg(4) = 1$, it takes $O(1)$ time to locate vertex 5 in $AList(4)$. Using the cross-pointer, it also takes $O(1)$ time to delete 4 in $AList(5)$. Since $Deg(5) = 2$, it is added to M.

In Figure 4.10, vertex 5 is selected from M. Vertex 2 is inserted into $AList'(6)$ while vertex 6 is inserted into $AList'(2)$. Since $col(2, 5)$ and $col(5, 6)$ are both cross,



M	AList	AList'
	1 : 2(crs) 6(crs)	1 :
4	2 : 1(crs) 3(crs)	2 :
2	3 : 2(crs) 6(crs) 5(crs)	3 :
1	4 : 5(crs)	4 :
	5 : 4(crs) 6(crs) 3(crs)	5 :
	6 : 1(crs) 5(crs) 3(crs)	6 :

Figure 4.8: Example of Implementation of Wiegers' Algorithm: a graph with 6 vertices



M	AList	AList'
	1 : 2(crs) 6(crs)	1 :
5	2 : 1(crs) 3(crs)	2 :
2	3 : 2(crs) 6(crs) 5(crs)	3 :
1	4 :	4 :
	5 : 6(crs) 3(crs)	5 :
	6 : 1(crs) 5(crs) 3(crs)	6 :

Figure 4.9: Example of Implementation of Wiegers' Algorithm: $u = 4$

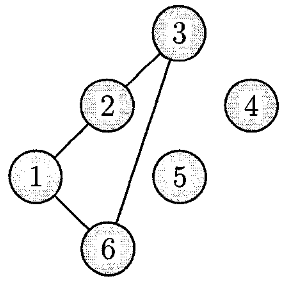
$col'(2,6)$ is thus assigned the color outer.

In next step (Figure 4.10), $Deg(3)$ becomes 2. Vertex 6 is removed from $AList'(3)$ and $AList(3)$ is search for an occurrence of vertex 6. Since $col(3,6)$ is cross, it is changed to outer. Since $Deg(3) = 2$, vertex 3 is returned to M (Figure 4.11).

Since $AList'(3)$ is empty, the two vertices adjacent to 3, namely 2 and 6, are removed from $AList(3)$. Since $col(2,3)$ is cross and $col(3,6)$ is outer, therefore $col'(2,6)$ is outer. Furthermore, vertex 2 is inserted into $AList'(6)$ while vertex 6 is inserted into $AList'(2)$. (Figure 4.12).

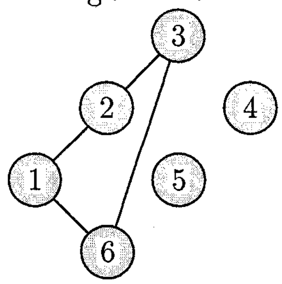
Similarly, vertex 1 is removed from M and the edge (2, 6) is added as in the previous step Figure 4.13.

Now, $AList'(2)$ and $AList'(6)$ are the only two lists that are non-empty. Furthermore, $AList'(2)$ contains two occurrences of 6 while $AList'(6)$ contains two



M	AList	AList'
	1 : 2(crs) 6(crs)	1 :
1	2 : 1(crs) 3(crs)	2 :
2	3 : 2(crs) 6(crs)	3 : 6(out)
6	4 :	4 :
3	5 :	5 :
	6 : 1(crs) 3(crs)	6 : 3(out)

Figure 4.10: Example of Implementation of Wiegers' Algorithm: $u = 5$

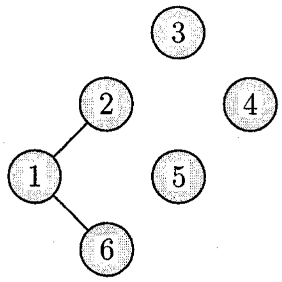


M	AList	AList'
	1 : 2(crs) 6(crs)	1 :
1	2 : 1(crs) 3(crs)	2 :
2	3 : 2(crs) 6(out)	3 :
6	4 :	4 :
3	5 :	5 :
	6 : 1(crs) 3(out)	6 :

Figure 4.11: Example of Implementation of Wiegers' Algorithm: $u = 3$

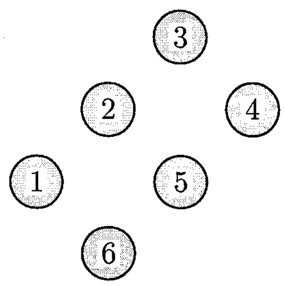
occurrences of 2. After the first occurrence of 2 and 6 are removed from the two lists, a new edge (2,6) is created and is given the color outer. After the second occurrence of 2 and 6 are removed from the two lists, as the edge (2,6) already exists, no new edge (2,6) is created. However, the color of (2,6) is changed to bridge (Figure 4.14).

Finally, after the edge (2, 6) is removed, the graph becomes edgeless. The input graph G is an outerplanar graph.



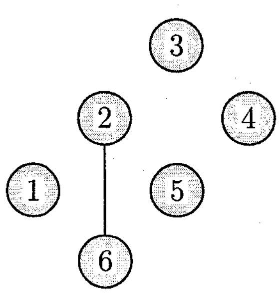
M	AList	AList'
1	1 : 2(crs) 6(crs)	1 :
2	2 : 1(crs)	2 : 6(out)
6	3 :	3 :
	4 :	4 :
	5 :	5 :
	6 : 1(crs)	6 : 2(out)

Figure 4.12: Example of Implementation of Wiegers' Algorithm: $u = 3$



M	AList	AList'
2	1 :	1 :
6	2 :	2 : 6(out) 6(out)
	3 :	3 :
	4 :	4 :
	5 :	5 :
	6 :	6 : 2(out) 2(out)

Figure 4.13: Example of Implementation of Wiegers' Algorithm: $u = 3$



M	AList	AList'
2	1 :	1 :
6	2 : 6(brg)	2 :
	3 :	3 :
	4 :	4 :
	5 :	5 :
	6 : 2(brg)	6 :

Figure 4.14: Example of Implementation of Wiegers' Algorithm: $u = 3$

Chapter 5

A Study of Tsin and Lin's Algorithm

In contrast with the algorithms of Michell and Wiegres, Tsin and Lin's [50] outerplanar graph algorithm is a DFS-based algorithm. The algorithm performs one DFS and does no sorting. During the DFS, the algorithm would abort its execution and output a "No" if a subgraph homeomorphic to K_4 or $K_{2,3}$ is detected; otherwise, it would terminate successfully with a "Yes" output. As with Wiegers' algorithm, this algorithm does not require the input graph to be biconnected.

5.1 Outerplanar algorithm

We shall first explain the idea underlying Tsin and Lin's algorithm.

A DFS is performed over the input graph to partition the graph into a collection of edge-disjoint paths such that every path contains exactly one back-edge. The paths are ordered using the following lexicographical order.

Definition 35. [50] *Let (q,p), (y,x) be two back edges such that $dfs(q) < dfs(p)$ and $dfs(y) < dfs(x)$. Then (q,p) is **lexicographically smaller** than (y,x), denoted by $(q,p) \prec (y,x)$, if and only if*

(i) $dfs(q) < dfs(y)$, or

(ii) $dfs(q) = dfs(y)$ and $dfs(p) < dfs(x)$ and p is not an ancestor of x, or

(iii) $dfs(q) = dfs(y)$ and $dfs(p) > dfs(x)$ and p is a descendant of x.

For each tree edge $(u, parent(u))$, we associate it with the back edge (y, x) with the smallest lexicographical rank such that x is a descendant of u and y is a proper ancestor of u. In this way, every tree edge is associated with a unique back edge. As a result, the edge set E is partitioned into a collection of subsets in which each subset contains exactly one back edge. It is easily verified that all the edges in the same subset form a path in G [50]. The following definitions are in order.

Definition 36. [50] *$Path_i$ is a path consisting of one back edge and all the tree edges associated with the back edge, where i is the rank of the back edge in lexicographical order.*

Definition 37. [50] *A path is a **non-trivial** path if it contains at least one tree edge. Otherwise, it is a **trivial** path.*

As there are a total of $|E| - |V| + 1$ back-edges, the collection of paths can be denoted by $\{path_i | 1 \leq i \leq |E| - |V| + 1\}$, where i is the rank (in lexicographical order) of the back edge that determine $Path_i$. Furthermore, $Path_1$ is always non-trial and is a cycle. Note that the non-trivial path are not generated explicitly. The are generated during the depth-first search.

Definition 38. [50] *A back edge (u, v) is an **incoming** (**outgoing**, respectively) back edge of u (v, respectively) if u is an ancestor of v.*

The algorithm is based on the following new characterization of outerplanar graph.

Theorem 7. *A graph is outerplanar if and only if all of the following conditions hold:*

i *with the exception of $Path_1$, the two end points of every non-trivial path are connected by a tree edge;*

ii *for every tree edge, there is at most one non-trivial path terminating at its two end points;*

iii *on every non-trivial path, no two there are two back edges interlace with each other.*

Proof: See [50]. □

A violation of either condition (i) or (ii) implies that the graph G contains a subgraph that is homeomorphic to $K_{2,3}$, while a violation of condition (iii) implies that the graph contains a subgraph that is homeomorphic to K_4.

The depth-first search starts at an arbitrary vertex r. During the depth-first search, the algorithm checks for a violation of any one of the three conditions stated in Theorem 7. If a violation is discovered, the algorithm would abort its execution immediately and output a "No" to indicate that the input graph is non-outerplanar. Otherwise, it would terminate its execution successfully and output a "Yes". The algorithm maintains the following variables for detecting violation of any of the three conditions:

Definition 39. [50] $\forall u \in V$, *$Path_u$ is the non-trivial path containing the tree edge $(u, parent(u))$, $Path1_u$ is a non-trivial path terminating at u and $parent(u)$.*

Definition 40. [50] $\forall u \in V$, *Z_u ($Z1_u$, respectively) is the vertex lying on $Path_u$ ($Path1_u$, respectively) such that (Z_u, u) ($(Z1_u, u)$, respectively) is the lexicographically largest incoming back edge of u.*

Definition 41. $\forall u \in V, lowpt(u) = \boldsymbol{min}(\{dfs(u)\} \cup \{lowpt(w) | w \text{ is a child of } u\} \cup \{dfs(s) | (u, s) \text{ is an outgoing back-edge of } u\})$;

When a vertex u is the current vertex of the depth-first search, the variables, $path_u$, $path1_u$, Z_u, $Z1_u$, $lowpt(u)$ and $dfs(u)$ (the depth-first search number of u, see Chapter 2) are defined for u.

Whenever the depth-first search backtracks from a child vertex, w, of u, if $Path_u$ already exists such that its two end-points are not connected by a tree-edge and $lowpt(w) < dfs(parent(u))$, then a violation of Condition (i) is detected; if $Path1_u$ already exists and $lowpt(w) = dfs(parent(u))$, then a violation of Condition (ii) is detected.

Whenever an outgoing back edge, (u, w), of u is encountered, if $dfs(w) < lowpt(u)$ and $Path_u$ is defined, then a violation of Condition (i) is discovered.

When the depth-first search backtracks from vertex u to its parent, if there is a vertex v lying on the path connecting u and Z_u ($Z1_u$, respectively) on $Path_u$ ($Path1_u$, respectively) such that v has an outgoing back edge (v,y) and $dfs(y) < dfs(u)$. Then a violation of Condition (iii) is detected.

A brief description of Tsin and Lin's algorithm is presented in Algorithm 12.

Algorithm 12 Tsin and Lin's Outerplanar Algorithm [50]

if ($|E| > 2|V| - 3$) **then**
  Output "No"
end if;
count $\leftarrow$ 1; **comment:** /* Initialize the counter for dfs number */
Outerplanar-testing($1, null, \perp$); **comment:** /*start DFS from vertex 1 */

5.2 An Example of Tsin and Lin's Outerplanar Algorithm

Figure 2.1 shows the depth-first search spanning tree created by a depth-first search. The number if the circles representing the vertices are the depth-first search numbers.

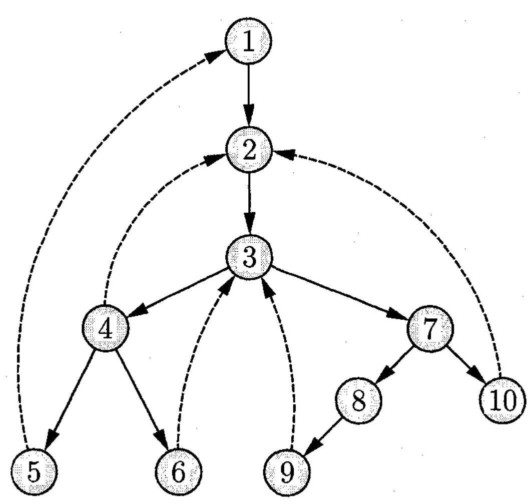


Figure 5.1: a DFS spanning tree of the graph in Figure 2.1

Figures 5.2, 5.4, 5.5, 5.6 depict the non-trivial paths P_1, P_3, P_4 and P_7, respectively. Note that P_1 is a cycle. When the depth-first search backtracks from

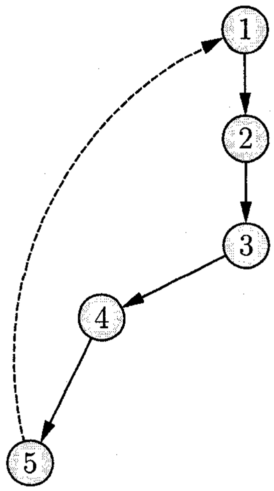


Figure 5.2: non-trivial path P_1

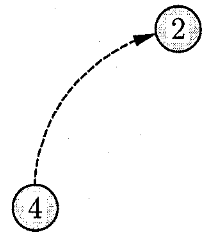


Figure 5.3: trivial path P_2

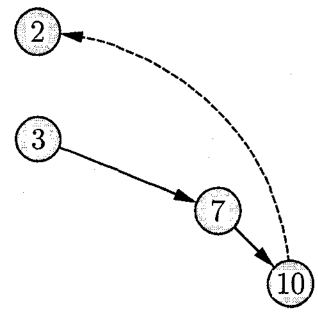


Figure 5.4: non-trivial path P_3

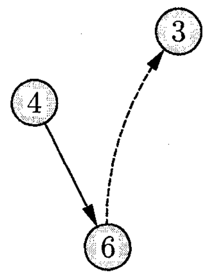


Figure 5.5: non-trivial path P_4

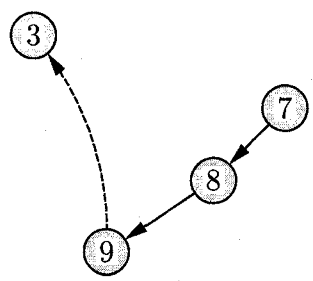


Figure 5.6: non-trivial path P_5

vertex 6 to vertex 4, since $lowpt(4) = 1$ and $lowpt(6) = dfs(parent(4))$, no violation of any condition is detected. When the depth-first search backtracks from vertex 8 to vertex 7, since $lowpt(7) = 2$ and $lowpt(8) = dfs(parent(7))$, again no violation of any condition is detected. When the depth-first search backtracks from vertex 7 to vertex 3, since $lowpt(3) = 1$ and $lowpt(7) = dfs(parent(3))$, again no violation of any condition is detected. At vertex 2, neither the back edge (2,4) nor the back edge (2,10) creates a situation that violates Condition (iii). The depth-first search thus terminates at the root 1 reporting that the graph is outerplanar.

During the DFS, whenever there is a non-trivial path whose terminating vertices are u and $parent(u)$, for some $u \in V$, the algorithm would mark this tree edge. If any tree edge is marked twice, then Condition (ii) is violated and the given graph is non-outerplanar. In the given example, this case does not happen.

5.3 Implementation

Tsin and Lin's algorithm is based on depth-first search which it is easy to implement, We shall thus refrain from explaining its implementation in this thesis. However, we shall remark that in our implementation, we did notice that recursive calls induced substantial run-time overhead. We thus replaced the recursive calls with iterations by explicitly maintaining the run-time stack that stores the current vertices of the depth-first search.

```
Procedure Outerplanar-testing(u, Path_u, v)
  dfs(u) ← count; count ← count + 1; lowpt(u) ← dfs(u); alert_u ← false;
  Path_u.type ← trivial; Path_u ← u; Z_u ← u; Path1_u ← u; Z1_u ← u;
  for each w in the adjacency list of u do
    if w is unvisited then
      if both Path_u and Path1_u have been found then return(false);
      Outerplanar-testing(w, Path_w, u);
      if (lowpt(w) < lowpt(u)) then
        if Path_u is non-trivial then
          if the end points of Path_u are not connected by a tree edge then
            return(false);
          else
            Path1_u ← Path_u; Z1_u ← Z_u; /* update Path1_u and Z1_u */
          end if
        else
          Label Path_u as non-trivial;
          if ((Path_w terminates at u and parent(u))) then mark the tree edge
          (u, parent(u));
          Path_u ← u || Path_w; Z_u ← u; lowpt(u) ← lowpt(w)
          /* || represents the concatenation operator for sequences */
        end if
      else
        if (lowpt(w) > lowpt(u)) then
          if (tree edge (u, parent(u)) has been marked) ∨ ( the two end points
          of Path_w are not connected by a tree edge ) then
            return(false)
          end if
          mark the tree edge (u, parent(u)); Path1_u ← u || Path_w; Z1_u ← u;
        else
          if (Path_u is non-trivial) then
            if ( v is not the root ∨ tree edge (u, parent(u)) has been marked)
            then
              return(false)
            else
              mark the tree edge (u, parent(u)); Path1_u ← u || Path_w
            end if
          else
            Path_u ← u || Path_w; Z_u ← u; Label Path_u as non-trivial;
          end if
        end if
      end if
    else
      backEdge(u, w);
    end if
  end for
  if (Z_u ≠ u) then bTest(Path_u, Z_u)
  if (Z1_u ≠ u) then bTest(Path1_u, Z1_u);
```

Procedure *bTest*($Path, Z$)

if ($\exists v$ connecting vertices u and Z on the path $Path$ such that v has an outgoing back edge (y, v) and $dfs(y) < dfs(u)$) **then**
return(false)

Procedure *backEdge*(u, w)

```
if (w, u) is an outgoing back edge of u then
  if (dfs(w) < lowpt(u)) then
    if Path_u is non-trivial then
      if Path_u is not terminating at u and parent(u) then
        return(false)
      end if
      Label Path_u as trivial;
      Path1_u ← Path_u; Z1_u ← Z_u; Path_u ← u; Z_u ← u
    end if
    lowpt(u) ← dfs(w)
  end if
else
  if ((w, u) is an incoming back edge of u) then
    if w lies on Path_u then
      if (dfs(w) > dfs(Z_u)) then Z_u ← w;
    else
      if w lies on Path1_u then
        if (dfs(w) > dfs(Z1_u)) then Z1_u ← w;
      end if
    end if
  end if
end if
```

Chapter 6

Experiments

We selected Mitchell's, Wiegers', Tsin and Lin's algorithms to implement and compare their behaviors using a total of 175 randomly generated graphs.

6.1 Experimental Data

6.1.1 The Input Graphs

In generating the input graphs, we take the following factors into consideration:

- Since all of the three algorithms terminates immediately if the input graph satisfies $|E| > 2|V| - 3$, therefore it is worth nothing to include those graphs in our experiment.

- Since Mitchell's algorithm only accepts biconnected graphs, all the input graphs generated are biconnected graphs.

- $|V|$ and $|E|$ are randomly generated. The possibility of an edge connecting two vertices is independent of the vertices themselves.

We randomly generated 175 simple graphs (graphs without self-loops and parallel edges). The 175 graphs consists of 85 non-outerplanar graphs and 90 outerplanar graphs. The number of vertices of the graphs ranges from 25,748 to 1,922,064, and the number of edges ranges from 25,926 to 3,799,671. Although it is desirable to generate more random graphs for our experiment, the performances of the three algorithms depicted in Figures 6.1, 6.2 and 6.3 clearly show the trend of the performance of each algorithm. Increasing the number of random graphs will not change the trends.

Biconnected Graphs

The algorithm for generating a random biconnected Graph $G = (V, E)$ is shown in Algorithm 13.

Algorithm 13 Random biconnected Graphs

Randomly generate $|V|$. Let $V = \{1, 2, ..., |V|\}$;
Connect 1 and 2, 2 and 3,..., $|V| - 1$ and $|V|$, $|V|$ and 1;
Randomly generate $|E|$ such that $|V| \leq |E| \leq (2|V| - 3)$;
for $i = |V|$ to $|E|$ **do**
  repeat
    Randomly select two vertices a, b;
  until an edge (a, b) has not been created before;
  Add edge (a, b) into the graph
end for

The Graph File

An adjacency list is used to represent the graph generated by Algorithm 13. The number of vertices and the number of edges are randomly generated. Each graph is stored in a binary file and is made up of three parts: the number of vertices, the number of edges and the edges denoted by two end vertices. The total size of graph files is around 2.85 Gbytes.

6.1.2 Experimental Results

We have conducted all the tests on operating system Fedora Core 4 which runs on Intel Pentium 4 2.60 Ghz processors and 512 Mbyte Memory. The programs are written in C. The execution time is reported in seconds, which is the user program CPU time, not including system CPU time. The performances of the algorithm are shown in the following figures.

In Figure 6.1, the performances of three algorithms on all the graphs are shown. Tsin and Lin's algorithm clearly has the best performance. The performances of Mitchell's and Wiegers' algorithms are close. However, when the number of edges goes beyong 1 million, Wiegers' algorithm begins to outperform that of Mitchel's.

In Figures 6.2 and 6.3, the performances of the three algorithms for outerplanar graphs and non-outerplanar graphs, respectively, are shown. For both

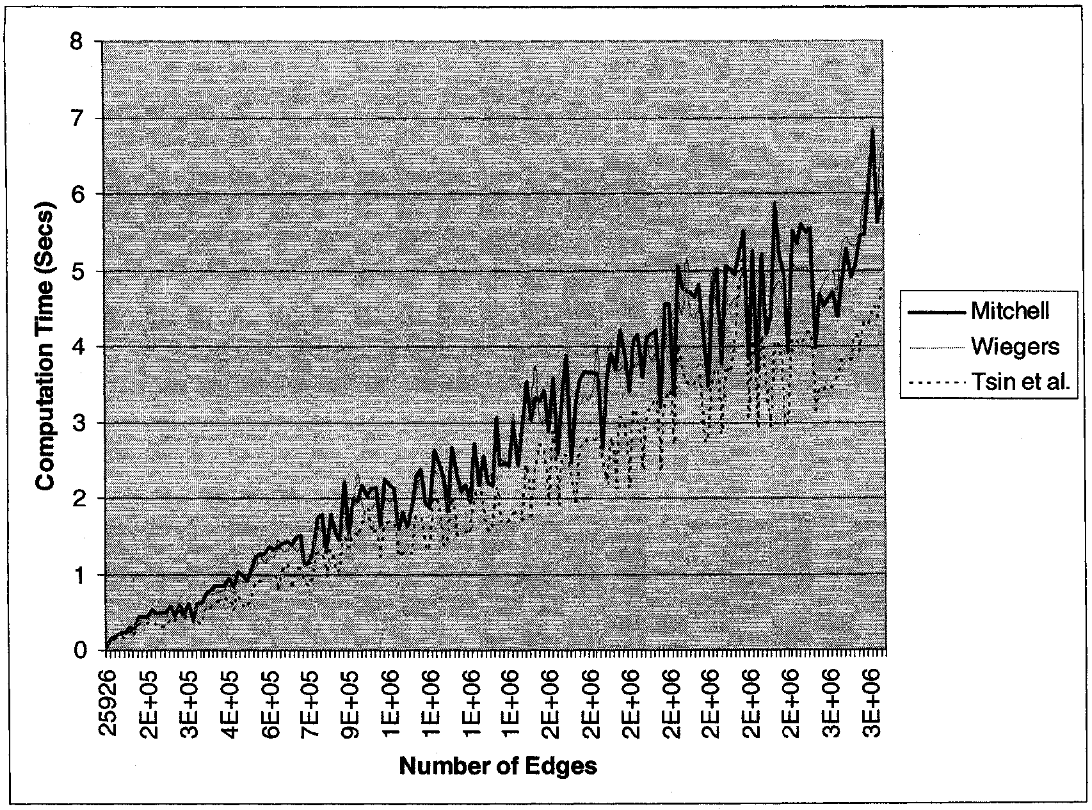


Figure 6.1: The performances of the three algorithms on all graphs, as a function of the graph size

groups of graphs, Tsin and Lin's algorithm has the best performance, especially for graphs with large edge sizes, the difference becomes more apparent. As shown in Figure 6.3, the performance of Mitchell's algorithm does not differ much with Wieger's when the input graph is non-outerplanar and has fewer than 2 million edges. However, when the edge size goes beyond 2 millions, Mitchell's algorithm has a better performance. On the other hand, for outerplanar graphs, Mitchell's algorithm is always the worst one (Figure 6.2).

6.2 Discussion

Tsin and Lin's algorithm is definitely the most efficient one in all cases. Between Mitchell's algorithm and Wiegers' algorithm, while Mitchell's has a better performance for non-outerplanar graphs, Wiegers' has a better perfomance for outerplanar graphs. This can be explained as follows: the bucket-sort used in

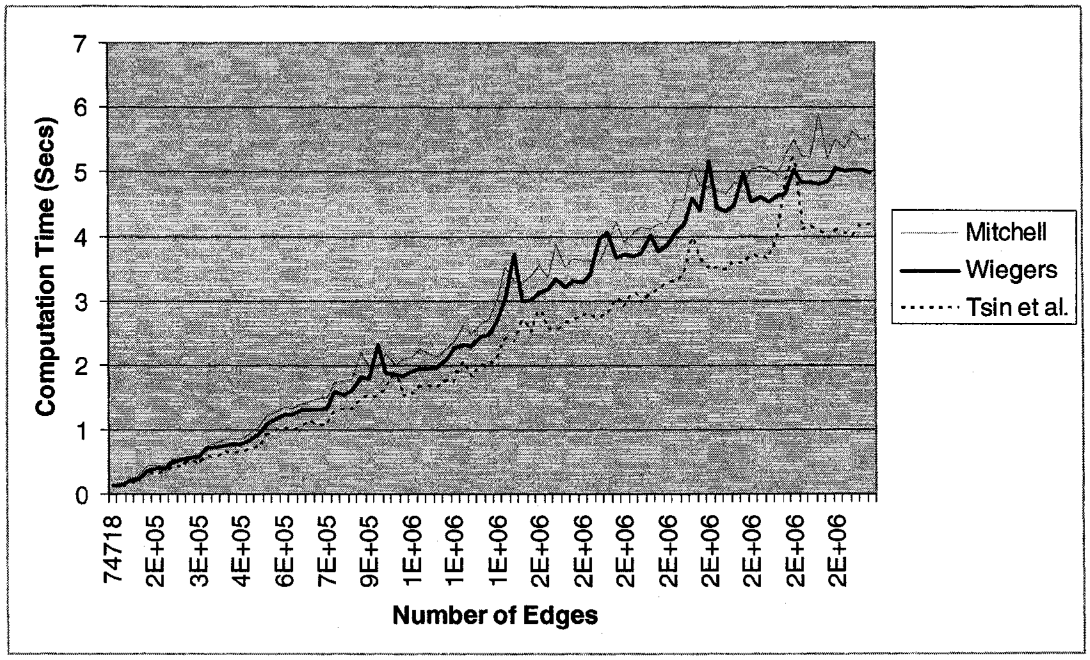


Figure 6.2: The performances of the three algorithms on Outerplanar Graphs, as a function of the graph size

Mitchell's algorithm is extremely time-consuming for larger input sizes. In dealing with non-outerplanar graphs, Mitchell's algorithm could terminate before doing bucket-sort. This allows it avoids doing the time-consuming sorting. For outerplanar graphs, bucket-sorting is an unavoidable step in Mitchell's algorithm.

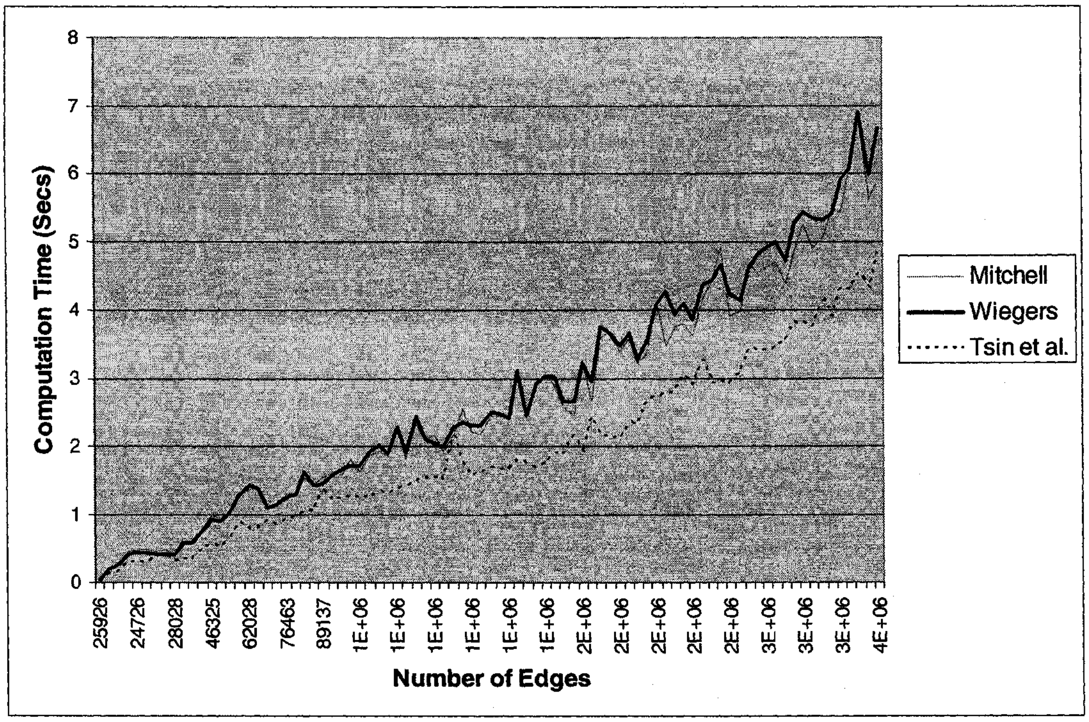


Figure 6.3: The performances of the three algorithms on non-Outerplanar Graphs, as a function of the graph size

Chapter 7

Embedding of Outerplanar Graphs

Once a graph is determined to be outerplaner, it is important to generate an outerplanar embedding for it. Of the three algorithms we have investigated, only Tsin *et al.* generates an outerplanar embedding in linear time and space. In this chapter, we shall modify Mitchell's algorithm so that it will generate an embedding for outerplanar input graph in linear time and space.

7.1 A Modified Mitchell's Algorithm for Outerplanar Embedding

We shall modify Mitchell's outerplanar algorithm so as to generate an outerplanar embedding for the input graph if the graph is outerplanar. The outerplanar embedding is represented by a sequence of the vertices along the boundary of the exterior face. The sequence is stored in a doubly-linked list $OuterList(u)$. Note that the two vertices preceding and following a vertex in the linked list are the two vertices adjacent to the vertex on the boundary of the exterior face.

We shall first briefly explain the idea underlying our modification.

Initially, all the vertices of degree two are inserted into a *queue* rather than a stack. We shall continue using $LIST$ to represent the queue. The reason of using a queue is that all the vertices that are of degree two initially have both incident edges lying on the boundary of the exterior face and hence should be dealt with first. To determine the boundary of the exterior face, it suffices to determine, for each vertex u, the two vertices (or two edges) on the boundary that are adjacent (or incident) to u. They are determined when the vertex u is removed from the queue $LIST$.

When a vertex u is removed from $LIST$, it is of degree 2 and hence must have exactly two incident edges (u, v) and (u, w), for some $v, w \in V$. We must determine if any of the two edges belongs to the boundary of the exterior face. Our method is to *mark* all the edges that do not lie on the boundary of the exterior face. These are exactly those edges that either are new edges added to the graph or appeared as $(NEAR, NEXT)$ in the course of executing the algorithm. As a result, we modify Mitchell's algorithm to mark these edge when they are created or encountered. The modified Mitchell's algorithm is presented as Algorithm 14. The new instructions are in bold-italic font. Some explanations are in order:

On Line 13, whenever a vertex of degree two is removed, Algorithm *AddEdgeto-Boundary* is called to include its unmarked incident edges to the linked list representing the boundary of the exterior face.

In Procedure *ChkAdj*, a and b are the two vertices adjacent to the most recently removed vertex. The edge connecting them cannot be an edge on the boundary of the exterior face and must thus be marked. Therefore, Procedure *ChkAdj* is modified as follows: If a and b are not connected by an edge, then an edge (a, b) is added to the graph and the edge is marked at both end points indicating that it is not an edge on the boundary. On the other hand, suppose a and b are connected by an edge. Then the white b in the adjacency list of a is marked. Moreover, if $|LIST| = 2$, then the white a in the adjacency list of b can be marked immediately. Otherwise, a red a is added to the adjacency list of b. This is to ensure that when the degree of vertex b is reduced to 2 and vertex b is removed from $LIST$, the red a will lead to the marking of edge (a, b) at vertex b.

When the adjacency list of a is scanned, every red vertex v in the list corresponds to an edge (a, v) that is not on the boundary and must thus marked at both end-points. If the edge (a, v) does not exist, then it is created by calling Procedure *AddWhite*(a, v) in which the edge is marked at both end points a and v. If the edge does exist, then it is marked at a. The edge will be marked at v later on when the adjacent list of v is scanned and a red a is encountered.

In Procedure *AddWhite*, whenever a new edge is added, the edge is marked at both end points to indicate that it does not lie on the boundary of the exterior face.

Procedure *AddEdgetoBoundary* adds unmarked edges incident to the vertex u (the most recent vertex removed from $LIST$) to the linked list representing the boundary of the exterior face.

Algorithm 14 Modified Mitchell's Outerplanar Algorithm

if $(|E| \leq 2|V| - 3)$ **then**
  Output "No"
end if;
$LIST \leftarrow \{v|Deg[v] = 2\}$; $PAIRS \leftarrow \emptyset$;
if $(|LIST| < 2)$ **then**
  Output "No"
end if;
for $L = 1$ **to** $|V| - 2$ **do**
  $NODE \leftarrow$ **front**$(LIST)$;
  $NEAR, NEXT \leftarrow$ the two vertices adjacent to $NODE$;
  Add $(NEAR, NEXT)$ to list $PAIRS$;
  Remove $NODE$ from the graph;
  Decrement $Deg(NEAR)$ and $Deg(NEXT)$;
  AddEdgetoBoundary$(NODE, NEAR, NEXT)$;
  if $(Deg(NEAR) \leq 2)$ **then**
    ChkAdj$(NEAR, NEXT)$;
  end if;
  if $(Deg(NEXT) \leq 2)$ **then**
    ChkAdj$(NEXT, NEAR)$;
  end if;
  if $(Deg(NEAR) > 2) \wedge (Deg(NEXT) > 2)$ **then**
    AddRed$(NEXT, NEAR)$;
  end if;
  if $(Deg(NEAR) \leq 2)$ **then** Add $NEAR$ to the end of $LIST$;
  if $(Deg(NEXT) \leq 2)$ **then** Add $NEXT$ to the end of $LIST$;
  if $(|LIST| - L < 2)$ **then**
    Output "No"
  end if
end for;
Add the edge $(NEAR, NEXT)$ to $EDGES$;
Lexicographically sort $EDGES$;
Lexicographically sort $PAIRS$;
if there is an edge in $PAIRS$ and not in $EDGES$ **then**
  Output "No"
else
  Output "Yes"
end if

Algorithm 15 Check the adjacency list of vertex a for vertex b

Procedure *ChkAdj(a,b)*

```
if (there is no b colored white in the adjacency list of a) then
  AddWhite(a, b)
else
  mark the white vertex b;
  if (|LIST| > 2) then
    add an a to the adjacency list of b; color the a red;
  else
    mark the white vertex a in the adjacency list of b;
  end if
end if;
for (each vertex v in the adjacency list of a) do
  if (Deg[v] = 0) then Remove v from the list;
  else if (v is red) then
          if (∄ another v colored white in the list) then
             RemoveRed((a, v));
             AddWhite(a, v);
          else RemoveRed(a, v);
               mark the white v;
```

Algorithm 16 Add White vertex

Procedure *AddWhite(a,b)*

```
Add the edge (a,b) to list EDGES;
Add a with white color to the end of the adjacency list of b; mark the a;
Add b with white color to the end of the adjacency list of a; mark the b;
Increment(Deg(a)); Increment(Deg(b));
```

Algorithm 17 *AddEdgetoBoundary* (u,v,w);

Comment: Add the edges (v,u) and (u,w) to the boundary
if (v in the adjacency list of u is unmarked) **then**
  Add v to $OuterList(u)$; Add u to $OuterList(v)$;
end if;
if (w in the adjacency list of u is unmarked) **then**
  Add w to $OuterList(u)$; Add u to $OuterList(w)$;
end if

7.2 Proof of Correctness

Theorem 8. *Let $G = (V, E)$ be a connected biconnected graph such that $|V| > 2$. Then $\forall u \in V, Deg(u) \geq 2$.*

Proof: Suppose to the contrary that $\exists u \in V$ such that $Deg(u) < 2$. Then $Deg(u) = 0$ or $Deg(u) = 1$. In the former case, G is disconnected. In the latter case, the vertex adjacent to u is a cut-vertex which implies that the graph G is not biconnected. In either case, we have a contradiction. □

Theorem 9. *Let $G = (V, E)$ be a biconnected outerplanar graph. Let $u \in V$ such that $Deg(u) = 2$ and v and w be the two vertices adjacent to u. Then the edges (v, u) and (u, w) lie on the boundary of the exterior face.*

Proof: For every vertex u in an outerplanar graph, there exist two adjacent vertices of u on the exterior face. Since $Deg(u) = 2$, the edges (v, u) and (u, w) are the only two edges incident to u. They must thus lie on the boundary of the exterior face. □

Theorem 10. *Let u be a vertex removed from LIST and (u, v) be an edge incident on it. The edge (u, v) is marked if and only if it does not lie on the boundary of the exterior face.*

Proof: First, note that all the edges are unmarked initially.

Suppose the edge (u, v) lies on the boundary of the exterior face. If $Deg(u) = 2$ originally, then vertex u is put into the queue $LIST$ at the beginning of the execution of the algorithm. Therefore, when vertex u is removed from $LIST$, the edge (u, v) remains as unmarked. If $Deg(u) > 2$ originally, then as only those edges that have appeared as $(NEAR, NEXT)$ during execution are marked, the edge (u, v) will never be marked as it will never appear as $(NEAR, NEXT)$ owing to the biconnectivity of the graph.

Suppose the edge (u, v) does not lie on the boundary of the exterior face. If (u, v) is created during execution, then it is marked immediately after its creation or is marked when the adjacency list of one of u or v is scanned at a later stage. On the other hand, if it exists in the original input graph, then it is marked either when it appears as the edge $(NEAR, NEXT)$ or at a later stage when the adjacency list of one of u or v is scanned. □

Theorem 11. *The modified Mitchell's Outerplanar algorithm correctly determines the boundary of the exterior face of an outerplanar graph.*

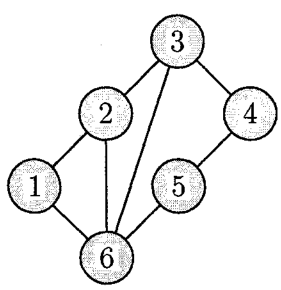
Proof. *Immediate from Theorems 8, 9, 10.* □

Theorem 12. *The modified Mitchell's Outerplanar algorithm takes $O(|V|)$ time and space to determine the boundary of the exterior face of an outerplanar graph.*

Proof: The new instructions increase the time and space complexity by a constant factor only. The theorem thus follows. □

7.3 An Example

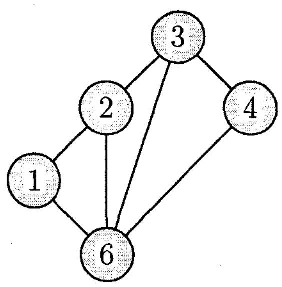
In this section, we give an example on how the modified Mitchell's Outerplanar algorithm produces an Outerplanar embedding for the graph in Figure 7.1.



OuterList	LIST	Adjacency List
1 :		1 : 2 6
2 :	5	2 : 1 6 3
3 :	4	3 : 2 4 6
4 :	1	4 : 3 5
5 :		5 : 4 6
6 :		6 : 1 2 5 3

Figure 7.1: Example of OuterPlanar Embedding (Mitchell's Algorithm)

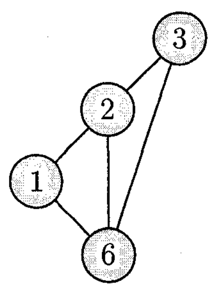
In Figure 7.1, the vertices 1, 4 and 5 are inserted into $LIST$ as these are the vertices that are of degree 2.



OuterList	LIST	Adjacency List
$\boxed{1}$:		$\boxed{1}$: 2 6
$\boxed{2}$:		$\boxed{2}$: 1 6 3
$\boxed{3}$:	4	$\boxed{3}$: 2 4 6
$\boxed{4}$: 5	1	$\boxed{4}$: 3 **6***
$\boxed{5}$: 4 6		$\boxed{6}$: 1 2 3 **4***
$\boxed{6}$: 5		

Figure 7.2: Example of OuterPlanar Embedding (Mitchell's Algorithm): after removal of vertex 5

In the next step, the vertex 5 is removed from $LIST$ Figure 7.2. The two adjacent vertices of 5, namely 4 and 6, are already stored in $OuterList(5)$. Vertices 4 and 6 are the two adjacent vertices of vertex 5. Since $Deg(4) = 2$, $AList(4)$ is examined. As the list does not contain a vertex 6, a marked vertex 6 is thus added at the end of $AList(4)$ while a marked vertex 4 is added to the end of $AList(6)$.



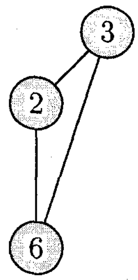
OuterList	LIST	Adjacency List
$\boxed{1}$:		$\boxed{1}$: 2 6
$\boxed{2}$:		$\boxed{2}$: 1 3 6
$\boxed{3}$: 4	1	$\boxed{3}$: 2 **6***
$\boxed{4}$: 3 5	3	$\boxed{6}$: 3_{red} 1 2 3
$\boxed{5}$: 4 6		
$\boxed{6}$: 5		

Figure 7.3: Example of OuterPlanar Embedding (Mitchell's Algorithm): after removal of vertex 4

In Figure 7.3, the removal of vertex 4 is similar to that of vertex 5. Vertex 6 in $AList(3)$ is marked and a vertex 3 with red color is added to $AList(6)$. Since $Deg(3) = 2$, vertex 3 is inserted into $LIST$.

Next, vertex 1 is removed from $LIST$. Vertices 2 and 6 are marked in $AList(6)$ and $AList(2)$, respectively. As $Deg(2) = 2$ and $Deg(6) = 2$, vertices 2 and 6 are inserted into $LIST$. Furthermore, the vertex 3 with red color is removed from $AList(6)$ and the white vertex 3 in $AList(6)$ is marked (Figure 7.4).

In the last step (Figure 7.5), vertex 3 is removed from $LIST$. $AList(3)$ is scanned and the unmarked vertex 2 is encountered. So vertex 2 is added to $OuterList(3)$ while vertex 3 is added to $OuterList(2)$.



OuterList	LIST	Adjacency List
$\boxed{1}$: 2 6		
$\boxed{2}$: 1	3	$\boxed{2}$: 3 **6***
$\boxed{3}$: 4	2	$\boxed{3}$: 2 **6***
$\boxed{4}$: 3 5	6	$\boxed{6}$: **2*** **3***
$\boxed{5}$: 4 6		
$\boxed{6}$: 5 1		

Figure 7.4: Example of OuterPlanar Embedding (Mitchell's Algorithm): after removal of vertex 1

OuterList	LIST	Adjacency List
$\boxed{1}$: 2 6		
$\boxed{2}$: 1 3		
$\boxed{3}$: 4 2	2	$\boxed{2}$: **6***
$\boxed{4}$: 3 5	6	$\boxed{6}$: **2***
$\boxed{5}$: 4 6		
$\boxed{6}$: 5 1		

Figure 7.5: Example of OuterPlanar Embedding (Mitchell's Algorithm): after removal of vertex 3

Now, $\forall u \in G$, $OuterList(u)$ are determined. An outerplanar embedding of the input graph is thus constructed.

7.4 A Modified Wiegers's Algorithm for Outerplanar Embedding

The modification for Wiegers' algorithm is quite simple. First, the input graph is decomposed into biconnected components. Next, an outplaner embedding for each of the biconnected components is determined. Finally, the outerplanar embeddings are joint at the cut-vertices to produce an outerplanar embedding for the input graph.

It remains to explain how to produce an outerplanar embedding for a biconnected graph.

In Wiegers' algorithm, edges are initially colored as *cross* edges. Therefore, if an edge is colored *cross* when it is removed from the graph, it must lie on the boundary of the exterior face.

An edge is colored *outer* if it is created during execution or it is converted from a *cross* edge. In the former case, it clearly cannot lie on the boundary of the exterior face. In the latter case, it cannot lie on the boundary of the exterior face unless it is the last edge left in the graph.

Since the graph is biconnected, an edge is colored *bridge* implies that it lies on two triangles. Therefore it cannot lie on the boundary of the exterior face.

The modification is clearly straight-forward and the resulting algorithm clearly takes linear time and space. The details are thus omitted.

Chapter 8

Conclusions

In this thesis, we presented the implementation of Mitchell's, Wiegers', Tsin and Lin's outerplanar graph algorithms. Mitchell's algorithm is based on a transformation of her maximal outerplanar graph algorithm. However, she only gave a brief description of the transformation and omitted many crucial details. Wiegers' algorithm briefly describes a $2-$reducible graph testing method and an edge-coloring technique, but did not point out how to implement them in linear time and space. We filled in all these non-trivial omitted details to clearly demonstrate how to implement them in linear time.

To the best of our knowledge, this is the first time a comparative study of the performances of outerplanar graph algorithms is carried out. The input graphs are randomly generated. The size of the input graph ranges from 25 thousands to 3.8 millions. Our experimental result shows that: Tsin and Lin's algorithm has the best performance among the three algorithms. Between Mitchell's and Wiegers' algorithms, Mitchell's has a better performance for non-outerplanar graphs while Wiegers' has a better performance for outerplanar graphs.

With the exception of Tsin and Lin's algorithm, Mitchell's and Wiegers' algorithms do not generate an outerplanar embedding if the input graph is indeed outerplanar. We presented a modification for each of the two algorithms so that an outerplanar embedding will be produced if the input graph is outerplanar. Correctness proofs of the modifications are presented. The complexity of the modified algorithms remain linear in both time and space.

It would be interesting to implement the outerplanar embedding algorithm of the aforementioned algorithms so that we could have better visualization of the input graph if it is outerplanar. This could be our future research.

Bibliography

[1] A.L.Buchsbaum, Goldwasser, M., Venkatasubramanian, S., and Westbrook, J. On external memory graph traversal. In *Procedding of the 11th ACM-SIAM Symposium on Discrete Algorithms* (2000), pp. 859–860.

[2] Arge, L., Meyer, U., Toma, L., and Zeh, N. On external-memoryin planar depth-first search. *Journal of Graph Algorithms and Applications 7*, 2 (2003), 105–129.

[3] Auslander, L., and Parter, S. On embeddings graphs in the plane. *J. Math. and Mech 10*, 3 (1961), 517–523.

[4] Awerbuch, B. A new distributed depth-first search algorithm. *Information Processing Letters 20* (1985), 147–150.

[5] Bachl, S. Isomorphic subgraphs. In *Proceedings Graph Drawing* (1999), Kratochvl, Ed., Springer, pp. 286–296.

[6] Booth, K., and Lueker, G. Testing the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Science 13* (1976), 335–379.

[7] Brehaut, W. An efficient outerplanarity algorithm. In *Procedding of the 8th Southeastern Conference on Combinatorics, Graph Theory and Computing* (Baton Rouge, Louisiana, February 1977), pp. 99–113.

[8] Chartrand, G., and Harary, F. Planar permutation graphs. *Ann. Inst. Henri Poincare, Sec B3 SE-9* (1967), 433–438.

[9] Cheung, T. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering SE-9*, 4 (1983), 504–511.

[10] Chiang, Y., Goodrich, M., Grove, E., Tamassia, R., Vengroff, D., and Vitter, J. External-memory graph algorithms. In *Procedding of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms* (1995), pp. 1–10.

[11] Cidon. Yet another distributed depth-first search algorithm. *Information Processing Letters 26* (1988), 301–305.

[12] COOK, S. The complexity of theorem-proving procedures. In *Theory of Computing* (1971), Proc. 3rd Annual ACM Symp., ACM, pp. 151–158.

[13] DANTZIG, G., AND FULKERSON, D. On the max–flow min–cut theorem of networks. In *Linear Inequalities and Related Systems* (Princeton, NJ, 1956), H. Kuhn and A. Tucker, Eds., 38, Annals of Mathematics Studies, Princeton University Press, pp. 215–221.

[14] DANTZIG, G., FULKERSON, D., AND JOHNSON, S. Solution of a large-scale traveling-salesman problems. *Oper. Res. Lett. 2* (1954), 393–410.

[15] DIJKSTRA, E. A note on two prlbems in connexion with graphs. *Numer. Math. 1* (1959), 269–271.

[16] DIKS, K., HAGERUP, T., AND RYTTER, W. Optimal parallel algorithms for the recognition and coloring outerplanar graphs. In *Proceedings of Math. Foundations of Computer Science* (Porabka-Kozubnik, Poland, 1989), pp. 207–217.

[17] E W. MYRVOLD, J. B. Stop minding your p's and q's: A simplified planar embedding algorithm. In *Proceddіng of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Maryland, January 1999), ACM Special Interest Group on Algorithms and Computation Theory and SIAM Activity Group on Discrete Mathematics, ACM Press, pp. 140–146.

[18] EDMONDS, J. Paths, trees and flowers. *Canada. J. Math 17* (1965), 449–467.

[19] EULER, L. The solution of a problem realint to the geometry of position. *Academy of Sciences of St. Petersburg* (1735).

[20] FREDERICKSON, G. Planar graph decomposition and all pairs shortest paths. *J.ACM 38* (1991), 162–204.

[21] FREDERICKSON, G., AND JANARDAN, R. Designing networks with compact routing tables. *Algorithmica 3* (1988), 171–190.

[22] FREDERICKSON, G., AND JANARDAN, R. Space-efficient and fault-tolerant message routing in outerplanar networks. *IEEE Transactions on Computers 37* (1988), 1529–1540.

[23] GOLDSTEIN, A. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. *Graph and Combinatorics Conference, Dept. Math., Princeton University* (1963), 16–18.

[24] GOMORY, R., AND HU, T. Multi-terminal network flows. *SIAM J. Appl. Math. 9* (1961), 551–556.

[25] GONÇALVES. Edge partition of planar sraphs into two outerplanar graphs. *Annual ACM Symposium on Theory of Computing* (2005), 504–512.

[26] GROSS, J., AND YELLEN, J., Eds. *Handbook of Graph Theory.* CRC Press, 2004.

[27] GUAN, M. Graphic programming using od or even points. *Chinese Math. 1* (1960), 273–277.

[28] HARARY, F. *Graph Theory.* Addison-Wesley, 1969.

[29] HARARY, F. *Graph Theory.* Addison-Wesley, 1994.

[30] HOPCROFT, J., AND TARJAN, R. Efficient planarity testing. *ACM 21*, 4 (1974), 549–568.

[31] HOPCROFT, J., AND TARJAN, R. Efficient planarity testing. *Journal of the Association for Computing Machinery 21*, 4 (1974), 549–568.

[32] KARP, R. Reducibility among combinatorial problems. In *Complexity of Computer Computations* (1972), R. Miller and J. Thatcher, Eds., Plenum Press, pp. 85–103.

[33] KAZMIERCZAK, A., AND RADHAKRISHNAN, S. An optimal distributed ear decomposition algorithm with applications to biconnectivity and outerplanar testing. *IEEE Transactions on Paralle and Distributed Systems 11* (2000), 110–118.

[34] KUMAR, D., IYENGAR, S., AND SHARMA, M. Correction to a distributed depth-first search algorithm. *Information Processing Letters 35* (1990), 55–56.

[35] LAKSHMANAN, K., MEENAKSHI, N., AND THULASIRAMAN, K. A time-optimal message-efficient distributed algorithm for depth-first search. *Information Processing Letters 25* (1987), 103–109.

[36] LEMPEL, A., EVEN, S., AND CEDERBAUM, I. An algorithm for planarity testing of graphs. *Proceddings International Symposium on Theory of Graphs* (1967), 215–232.

[37] MAHESHWARI, A., AND ZEH, N. I/o efficient algorithms for outerplanar graphs. *Journal of Graph Algorithms and Applications 8*, 1 (2004), 47–87.

[38] MAKKI, S., AND HAVAS, G. Distributed algorithms for depth-first search. *Information Processing Letters 60* (1996), 7–12.

[39] MARX, D. Np-completeness of list coloring and precoloring extension on the edge of planar graphs. *Journal of Graph Theory 49*, 4 (2005), 313–324.

[40] MITCHELL, S. Linear algorithm to recognize ouerplanar and maximal outerplanar graphs. *Information Processing Letters 9*, 5 (1979), 229–232.

[41] MITCHELL, S., BEYER, T., AND JONES, W. Linear algorithms for isomorphism of maximal outerplanar graphs. *Journal of the ACM 26*, 4 (1979), 603–610.

[42] PADBERG, M., AND RINALDI, G. Optimization of a 532–city symmetric traveling salesman problem by branch and cut. *Oper. Res. Lett.* (1987), 1–7.

[43] PROSKUROWSKI, A., AND SYSLO, M. Minimum dominating cycles in outerplanar graphs. *International Journal of Parallel Programming 10*, 2 (1981), 127–139.

[44] REIF, J. Depth first search is inherently sequential. *Information Processing Letters 20* (1985), 229–234.

[45] SHARMA, M., IYENGAR, S., AND MANDYAM, N. An efficient distributed depth-first search algorithm. *Information Processing Letters 32* (1989), 183–186.

[46] SHIH, W., AND HSU, W. A new planarity test. *Theoretical Computer Science 223* (1999), 179–191.

[47] SYSLO, M., AND IRI, M. Efficient outerplanarity testing. *Annales Societatis Mathematicae Polonae Series IV: Fundamenta Informaticae II* (1979), 261–275.

[48] TARJAN, R. Depth-first search and linear graph algorithms. *SIAM J.COMPUT 1* (1972), 146–160.

[49] TSIN, Y. Some remarks on distributed depth-first search. *Information Processing Letters 82* (2002), 173–178.

[50] TSIN, Y., AND LIN, Y. *On testing and embedding outerplanar graphs in linear time and space, Technical Report TR 04-022.* School of Computer Science, University of Windsor, 2004.

[51] TURAU, V. Computing bridges, articulations and 2-connected components in wireless sensor networks. In *ALGOSENSORS 2006, LNCS 4240* (2006), pp. 164–175.

[52] VON CHRISTINA WITWER. *Prediction of Conderved and Consensus RNA Structures, Ph.D. thesis.* Fakultat fur Naturwissenschaften und Mathematik der Universitat Wien, 2003.

[53] WIEGERS, M. Recognizing outerplanar graphs in linear time. In *International Workshop WG '86 on Graph-theoretic concepts in computer science* (1987), Springer-Verlag, pp. 165–176.

VITA AUCTORIS

NAME:	Tao Deng
PLACE OF BIRTH:	Chengdu, China
YEAR OF BIRTH:	1981
EDUCATION:	University of Electronic Science and Technology of China Chengdu, China 1999-2003
	University of Windsor, Windsor, Ontario 2005-2007