M.Sc. Engg. Thesis

Efficient Algorithms for Generating All Triangulations of Plane Graphs

by

Mohammad Tanvir Parvez

Submitted to

Department of Computer Science and Engineering
in partial fulfilment of the requirments for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka 1000

March 2006

The thesis titled "**Efficient Algorithms for Generating All Triangulations of Plane Graphs**," submitted by Mohammad Tanvir Parvez, Roll No. 040405013P, Session April 2004, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on March 8, 2006.

Board of Examiners

1. ______________________ Dr. Md. Saidur Rahman Associate Professor Department of CSE BUET, Dhaka 1000	Chairman (Supervisor)
2. ______________________ Dr. Muhammad Masroor Ali Professor & Head Department of CSE BUET, Dhaka 1000	Member (Ex-officio)
3. ______________________ Dr. Md. Abul Kashem Mia Professor Department of CSE BUET, Dhaka 1000	Member
4. ______________________ Dr. Masud Hasan Assistant Professor Department of CSE BUET, Dhaka 1000	Member
5. ______________________ Dr. Md. Elias Associate Professor Department of Mathematics BUET, Dhaka 1000	Member (External)

Candidate's Declaration

It is hereby declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Tanvir Parvez

Mohammad Tanvir Parvez
Candidate

Contents

Board of Examiners i

Candidate's Declaration ii

Acknowledgements ix

Abstract x

1 Introduction 1

1.1 Problem Statement 3

1.2 Applications 3

1.2.1 Graph Drawing 4

1.2.2 VLSI Floorplanning 7

1.2.3 Computational Geometry 8

1.3 Challenges 9

1.3.1 Time Complexity 10

1.3.2 Avoiding Duplications 10

1.3.3 I/O Operations 10

1.3.4 Exhaustive Generation 11

1.4 Goals of an Enumeration Algorithm 11

1.5 Literature Review 12

1.5.1 Algorithm of Hurtado and Noy 13

1.5.2 Algorithm of Li and Nakano 14
1.6 Scope of this Thesis 15
1.6.1 Labeled Triangulations of a Convex Polygon 16
1.6.2 Triangulations of a Biconnected Plane Graph 16
1.6.3 Unlabeled Triangulations of a Convex Polygon 16
1.7 Summary 16

2 Preliminaries **18**
2.1 Basic Terminology 18
2.1.1 Polygon 18
2.1.2 Graphs 19
2.1.3 Planar graph and plane graph 20
2.1.4 Triangulations of a Polygon 20
2.1.5 Labeled and Unlabeled Triangulations 22
2.1.6 Triangulations of a Graph 23
2.1.7 Graphs and Polygons 24
2.1.8 Paths and Cycles 24
2.1.9 Outerplanar Graph 24
2.1.10 Trees 24
2.1.11 Binary Trees 26
2.1.12 Degree Sequence 26
2.2 Algorithms and Complexity 26
2.2.1 The notation $O(n)$ 27
2.2.2 Polynomial algorithms 27
2.2.3 Complexity of Graph Algorithms 28
2.3 Graph Traversal Algorithm 28
2.4 Face Traversal 29
2.4.1 Data Structures for a Plane Graph 29

2.4.2 Face Traversal Algorithm 30
2.5 Algorithms for Enumeration Problems 31
2.5.1 Combinatorial Gray Code Approach 32
2.5.2 Family Tree Approach 34
2.6 Catalan Families 34

3 Triangulations of Convex Polygons **36**
3.1 Introduction 36
3.2 Basic Idea 37
3.3 Labeled Triangulations of a Convex Polygon 38
3.3.1 Child-Parent Relationship 41
3.3.2 Generating the Children of a Triangulation in $\mathcal{T}(P)$ 43
3.3.3 The Representation of a Triangulation in $\mathcal{T}(P)$ 48
3.3.4 The Algorithm 49
3.4 Unlabeled Triangulations of a Convex Polygon 53
3.4.1 Removing Rotational Repetitions 55
3.4.2 Avoiding Mirror Repetitions 58
3.5 Conclusion 59

4 Triangulations of Plane Graphs **60**
4.1 Introduction 60
4.2 Algorithm for Biconnected Outerplanar Graphs 61
4.2.1 Finding the Root 62
4.2.2 The Child Generation Rule 64
4.3 Algorithm for Biconnected Plane Graphs 65
4.4 Conclusion 65

5 Conclusion **67**

List of Figures

1.1 Triangulation of a plane graph by adding edges. 3
1.2 Illustration of the straight line grid drawing algorithm. 5
1.3 Illustration of the straight line grid drawing algorithm. The graph is triangulated in a different way than in Figure 1.2 6
1.4 Illustration of the algorithm for finding rectangular floorplan. 7
1.5 Illustration of the algorithm for finding rectangular floorplan. The graph is triangulated in a different way than in Figure 1.4 8
1.6 Illustration of the Art Gallery Problem 9
1.7 Graph of triangulations of convex polygons of six vertices. 13
1.8 Illustration of tree of triangulations of convex polygons. 14
1.9 Illustration of splitting operation. 14
1.10 Illustration of the generation of new triangulations from an old one. 15
1.11 Illustration of the tree of triangulations of biconnected based plane triangulations. 15

2.1 Illustration of a polygon. 19
2.2 Illustration of a graph. 20
2.3 Illustration of planar embedding. 21
2.4 Illustration of triangulated polygons (a) triangulation of a simple polygon and (b) triangulation of a convex polygon. 21
2.5 Two ways of triangulating a convex polygon of 6 vertices. 22

2.6 Illustration of labeled triangulations. 22
2.7 Illustration of unlabeled triangulations. 23
2.8 Illustration of a triangulated graph. 23
2.9 Example of (a) an outerplanar graph and (b) biconnected outerplanar graph. 25
2.10 Illustration of a tree. 25
2.11 Illustration of a binary tree. 26
2.12 Adjacency list representation of a plane graph. 30
2.13 Illustration of a data structure for representing a plane graph used for efficient face traversal. 31
2.14 Generating permutations using gray code approach: Johnson-Trotter scheme. 33
2.15 Illustration of relationship between triangulations and binary trees. 35

3.1 Illustration of (a) labeled and (b) unlabeled triangulations. 37
3.2 Genealogical tree $\mathcal{T}(P)$ for a convex polygon P of six vertices. 39
3.3 Illustration of flipping operation; (a) old triangulation and (b) new triangulation. 39
3.4 Illustration of (a) a convex polygon of six vertices and (b) corresponding root in $\mathcal{T}(P)$. 40
3.5 Illustration of generation of blocking diagonal; (a) old triangulation and (b) new triangulation. 41
3.6 Illustration of child-parent relationship; (a) child and (b) parent. 42
3.7 Illustration of a flipping operation that does not preserve parent-child relationship. 44
3.8 Illustration of generating diagonals. 44
3.9 Illustration of Lemma 3.3.4. 46
3.10 Illustration of Lemma 3.3.5. 47
3.11 Illustration of generation of a blocking diagonal from a generating diagonal 49

3.12 Illustration of the effects of a flipping operation 52

3.13 Illustration of update operations for opposite pairs of two affected edges . . 53

3.14 Illustration of rotationally equivalent triangulations 54

3.15 Illustration of triangulations which are mirror image of each other 54

3.16 Illustration of genealogical tree of six vertices when rotational and mirror repetitions are not allowed. 55

3.17 Illustration of two triangulations where one has greater degree sequence. . 56

4.1 Illustration of (a) an outerplanar graph G of eight vertices and (b) genealogical tree $\mathcal{T}(G)$ of G. 61

4.2 An inner face of a biconnected outerplanar graph can be viewed as a convex polygon. 62

4.3 Illustration of (a) a biconnected outerplanar graph of 10 vertices and (b) corresponding root in $\mathcal{T}(G)$ 63

4.4 Triangulating outer face of a plane graph 66

Acknowledgments

All praises due to Allah, Lord of the Worlds, who granted me the ability to finish this thesis.

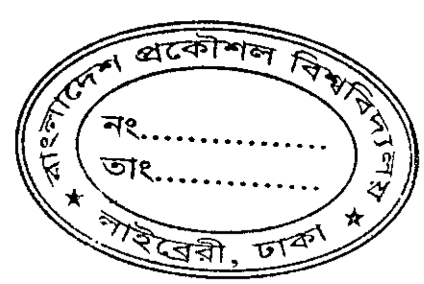
I would like to thank my supervisor Dr. Md. Saidur Rahman for introducing me to the field of enumeration of combinatorial objects. I have learned from him how to write, speak and present well. I thank him for his patience in reviewing my so many inferior drafts, for correcting my proofs and language, suggesting new ways of thinking and encouraging me to continue my work.

I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Professor Dr. Muhammad Masroor Ali, Professor Dr. Md. Abul Kashem Mia, Dr. Masud Hasan and Dr. Md. Elias.

It would be inappropriate if I do not mention the members of our research group. They gave me valuable suggestions and listened to all of my presentations. I just want to say them: Thank You!

Abstract

Generating all triangulations of graphs and polygons have many applications in Computational Geometry, VLSI Floorplaning and Graph Drawing. In this thesis, we deal with the problem of generating all triangulations of plane graphs. We give an algorithm for generating all triangulations of a biconnected plane graph G of n vertices. Our algorithm establishes a tree structure among the triangulations of G, called the "tree of triangulations," and generates each triangulation of G in $O(1)$ time. The algorithm uses $O(n)$ space and generates all triangulations of G without duplications. To the best of our knowledge, our algorithm is the first algorithm for generating all triangulations of a biconnected plane graph; although there exist algorithms for generating triangulated graphs with certain properties. Our algorithm for generating all triangulations of a plane graph needs to find all triangulations of a convex polygon. We give an algorithm to generate all triangulations of a convex polygon P of n vertices in time $O(1)$ per triangulation, where the vertices of P are numbered. Our algorithm for generating all triangulations of a convex polygon also improves previous results; existing algorithms need to generate all triangulations of convex polygons of less than n vertices before generating the triangulations of a convex polygon of n vertices. Finally, we give an algorithm for generating all triangulations of a convex polygon P of n vertices in time $O(n^2)$ per triangulation, where vertices of P are not numbered.


বাংলাদেশ প্রকৌশল বিশ্ববিদ্যালয় লাইব্রেরী, ঢাকা
নং...............
তাং...............

Chapter 1

Introduction

One of the problems addressed in the area of combinatorial algorithms is to generate all items of a particular combinatorial class in such a way that each item is generated exactly once. To solve many practical problems it is required to generate samples of random objects from a combinatorial class. Sometimes a list of objects in a particular class is helpful to find a counter-example to some conjecture, to find the best object among all candidates, or to experimentally measure the average performance of an algorithm over all possible inputs. Early works in combinatorics focused on counting; because generating all objects requires huge computation. With the aid of fast computers it now has become feasible to list the objects in combinatorial classes. However, in order to generate entire list of objects from a class of moderate size, extremely efficient algorithms are required even with the fastest computers. Due to the reason mentioned above, recently many researchers have concentrated their attention for developing efficient algorithms to generate all objects of a particular class without repetitions [JWW80, Sav97]. Examples of such exhaustive generation of combinatorial objects include enumerating all binary trees, generating all set partitions, generating permutations and combinations, enumerating spanning trees etc [BV04, NU04a, NU04b].

One of the most important and extremely useful class of combinatorial objects are

graphs. In this thesis, we deal with the problem of generating all triangulations of plane graphs. Generating all triangulations of plane graphs have many applications in Computational Geometry [DVOS00], VLSI floorplanning [SY99], and Graph Drawing [NR04]. In these applications, to get a better solution, sometimes it is necessary to find all triangulations of plane graphs. In this thesis, we also deal with problem of enumerating all triangulations of convex polygons. Polygon triangulation plays a central role in Computational Geometry and is a basic step in many algorithms [DVOS00]. But, developing algorithms for enumerating such triangulations is not an easy problem. The possible number of triangulations of plane graphs is exponential and for this reason, the enumeration algorithms would have to produce huge amounts of output. Such algorithms are I/O intensive, need huge computational power and thus require fast computers. So, any algorithm for enumerating triangulations of plane graphs has to be extremely efficient. However, due to the exponential number of possible outputs, any enumeration algorithm for generating all triangulations of plane graphs can be at best exponential. Therefore, such algorithms concentrate on the complexity of the individual object generation, rather than the complexity of the overall running time of the algorithm. In this thesis, we give algorithms for generating all triangulations of plane graphs and exploit clever algorithmic techniques to successfully meet the above mentioned challenges .

This chapter serves as an introduction to the problem we dealt with in this thesis. We also discuss related applications and review the literature. We start in Section 1.1 by giving a precise description of the problem we solved in this thesis. Section 1.2 describes some of the applications of the algorithms we developed in this thesis. Section 1.3 addresses the algorithmic challenges that any efficient enumeration algorithm must resolve. Section 1.5 discusses two problems related to the problem we have solved in this thesis and describes existing algorithms for those problems. Section 1.6 deals with the scopes of this thesis and finally Section 1.7 gives a summary of the results we have found and compares our algorithms with other related algorithms.

1.1 Problem Statement

In this thesis we deal with the problem of generating all triangulations of plane graphs. We give a brief description of the problem below.

Let $G = (V, E)$ be a plane graph as shown in Figure 1.1(a). The graph in Figure 1.1(a) has 4 faces; three of them are internal and the other one is external. If we add an edge (v_3, v_7) to the graph of Figure 1.1(a), we get the graph G' of Figure 1.1(b). As can be seen, the face $F_1 = v_2, v_3, v_6, v_7$ of the graph in Figure 1.1(a) has become "triangulated" in Figure 1.1(b); that is the face F_1 of the graph in Figure 1.1(a) has been partitioned into smaller faces in the graph of Figure 1.1(b) where each of the smaller faces of G' contains three edges on the boundary of it. We say that we have *triangulated* the face F_1 of G. By adding edges in G, we can triangulate all the faces of G. When all the faces of G are triangulated we get a triangulation of the graph G. Figure 1.1(c) shows one possible triangulation of G. A particular plane graph G may have many different triangulations. In this thesis we address the problem of generating all triangulations of a given plane graph G such that all the triangulations of G are generated without duplications.

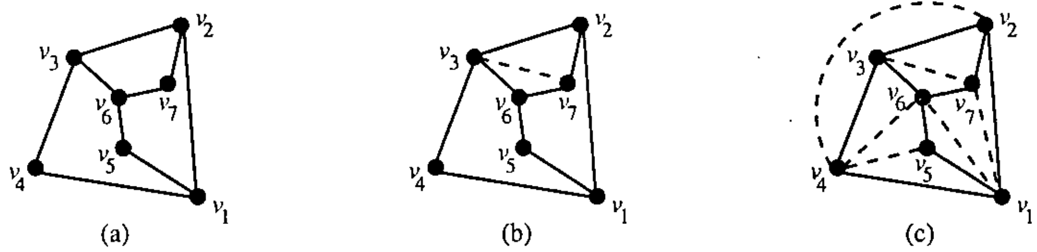


Figure 1.1: Triangulation of a plane graph by adding edges.

1.2 Applications

Our algorithms for generating all triangulations of plane graphs and polygons have a number of useful applications in a variety of fields. In this section we describe three different application domains where the algorithms for generating all triangulations of a

given plane graph or a simple polygon can be used.

1.2.1 Graph Drawing

Many algorithms in the area of Graph Drawing take triangulated plane graphs as input. If the input graph is not triangulated then we must find a suitable triangulation of the input graph before using the algorithm. For many algorithms, the way the input graph is triangulated affects the quality of the output of the algorithm. For example, one of the well known algorithms [NR04] for finding straight line grid drawing of a plane graph requires that the input graph G must be triangulated. If the graph G is not a triangulated already, then we must find a triangulation of G. The algorithm finds a canonical ordering of the vertices of G based on the triangulation of G and draws the straight line grid drawing of the graph using that ordering. The area required by a straight line grid drawing of a graph G depends on the canonical ordering of the vertices of G, which in turn depends upon the way G is triangulated. Two different triangulations of the same graph G may result in two different drawings of G with different area requirements. Figure 1.2 illustrates the algorithm for finding the straight line grid drawing for the graph of Figure 1.2(a). Figure 1.2(b) is a triangulation of the graph in Figure 1.2(a). In Figure 1.2(b), the numbers shown in parenthesis besides the labels of the vertices show the canonical ordering of the vertices of the graph. As shown in Figure 1.2(d), the straight line grid drawing of the graph of Figure 1.2(a) requires an 8×4 grid for the particular triangulation of the graph as shown in Figure 1.2(b). Whereas the same graph of Figure 1.2(a), triangulated in a different way, may have a grid drawing of different grid size, as shown in Figure 1.3.

Therefore, if we want to find a straight line grid drawing of a plane graph G which satisfies some area requirements, we may need to find all the triangulations of G. For example, if we want the straight line grid drawing of G which requires minimum amount of area, then we need an algorithm that generates all the triangulations of a given plane graph.

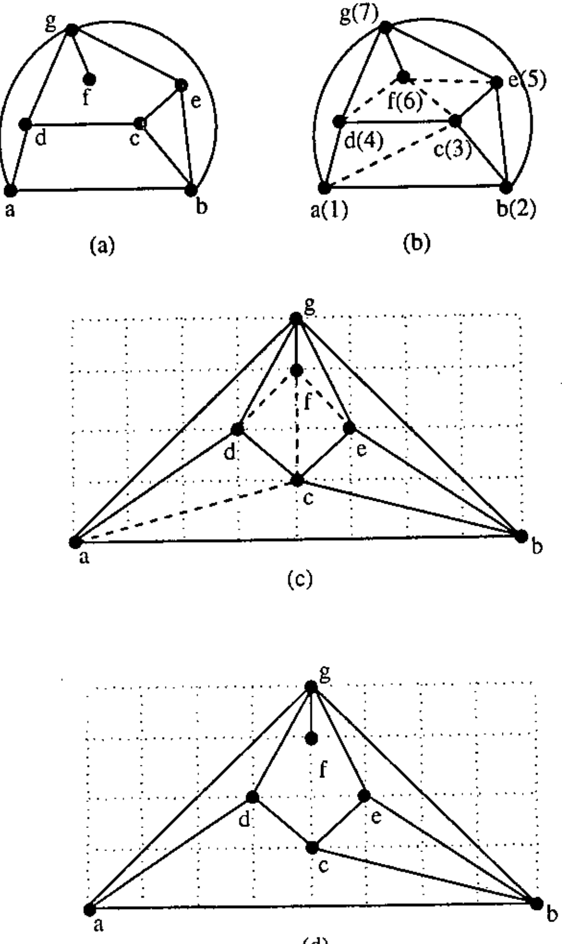


Figure 1.2: Illustration of the straight line grid drawing algorithm.

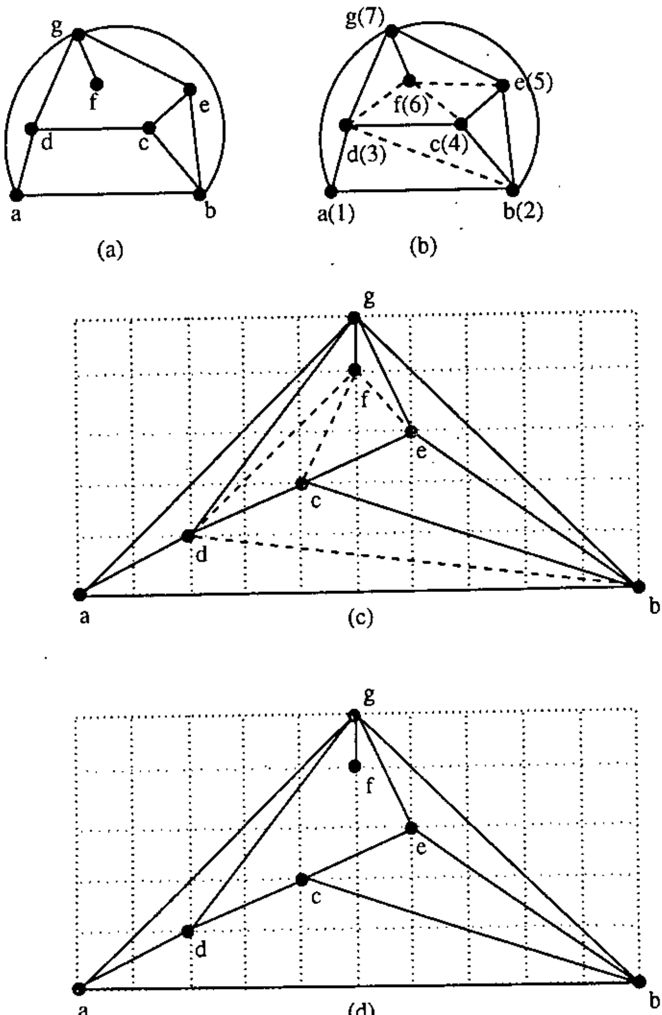


Figure 1.3: Illustration of the straight line grid drawing algorithm. The graph is triangulated in a different way than in Figure 1.2

1.2.2 VLSI Floorplanning

Our algorithm for generating all triangulations of a plane graph has useful applications in the area of VLSI floorplanning. One of the problems in VLSI floorplanning is to find a rectangular floorplan of a given plane graph G. Conventional floorplanning algorithms solve that problem by first triangulating the graph G, then finding a dual like graph G^* of G and then from G^* a rectangular floorplan of G is found [NR04, SY99]. For example, in Figure 1.4, to find a rectangular floorplan of the plane graph G of Figure 1.4(a), we need to find a triangulation of G as in Figure 1.4(b). Figure 1.4(c) is the dual like graph G^* of G of Figure 1.4(a) and from G^* we find a rectangular floorplan of G.

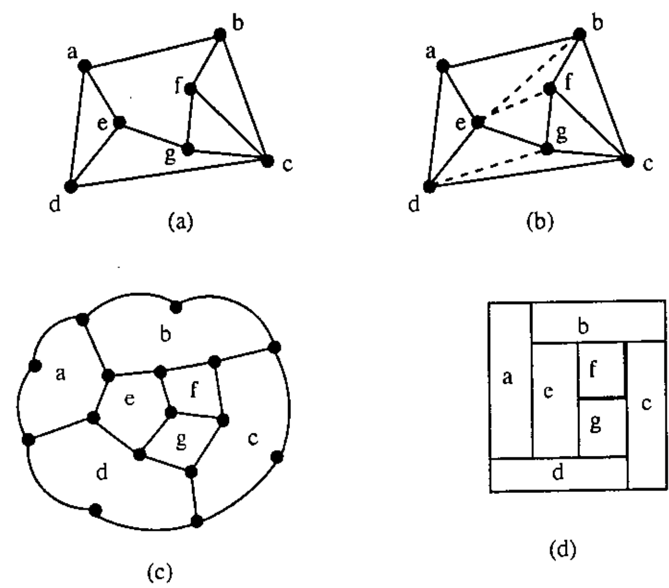


Figure 1.4: Illustration of the algorithm for finding rectangular floorplan.

An interesting observation is that if we triangulate the same plane graph G in a different way, we may get a different floorplan of G. An example of that is shown in Figure 1.5. The plane graph G of Figure 1.5(a) is the same graph of Figure 1.4(a), but triangulated in a different way as shown in Figure 1.5(b). As can be seen in Figure 1.5(d), the floorplan of G is somewhat different than the floorplan of Figure 1.4(a). Sometimes we may need to explore different floorplans of a plane graph G. For example, we may

want a particular floorplan that satisfies some aesthetic criteria or has some adjacency requirements for the faces. In these cases, we need to find all the different floorplans of a given plane graph G, which requires to have an algorithm that generates all the triangulations of a given plane graph.

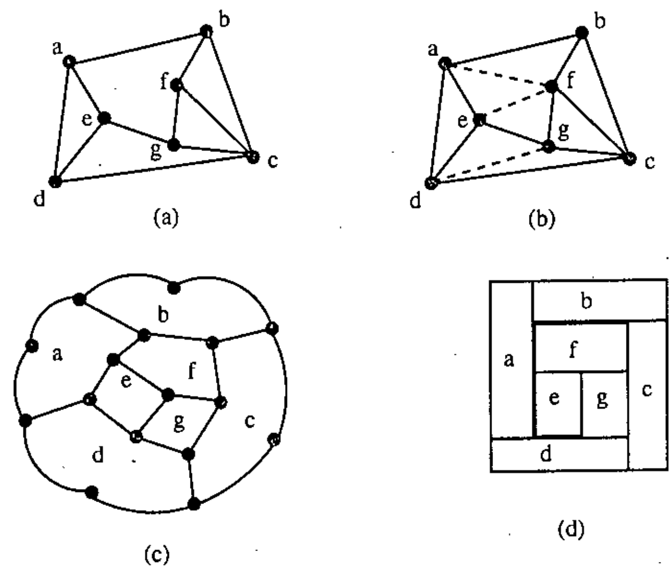


Figure 1.5: Illustration of the algorithm for finding rectangular floorplan. The graph is triangulated in a different way than in Figure 1.4

1.2.3 Computational Geometry

Several applications of triangulations are found in Computational Geometry. Polygon triangulation is vital in many of the algorithms in Computational Geometry. There exist different algorithms that can triangulate any simple polygon [DVOS00, Cha91]. One of the well known problems in Computational Geometry is the so called "Art Gallery Problem". In Art Gallery Problem, each floor of an art gallery is modeled as a simple polygon. The problem is to place cameras at the polygonal vertices in such a way that the entire floor of the art gallery is covered. Conventional algorithm for solving the Art Gallery Problem finds a triangulation of the simple polygon, colors the vertices of the triangulated polygon

with three colors such that adjacent vertices have different colors and then chooses the color which is used minimally. Cameras are placed at the vertices which are colored with that minimally used color. The number of cameras required depends on the way the simple polygon is triangulated. Different triangulations of the same simple polygon may give different results to the same Art Gallery Problem. Figure 1.6 illustrates such a case. In Figure 1.6, R, G and B stand for the colors Red, Green and Blue. The triangulation of Figure 1.6(a) gives a solution that requires three cameras; whereas the triangulation of Figure 1.6(b) requires only two cameras.

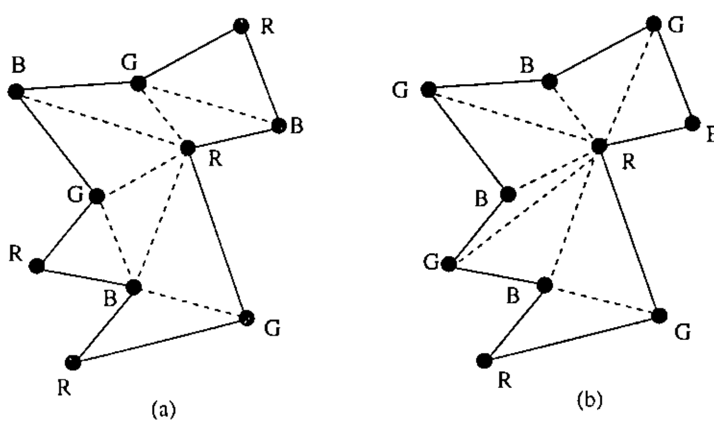


Figure 1.6: Illustration of the Art Gallery Problem. (a) Solution of size three and (b) solution of size two.

Finding the minimum number of cameras for a particular Art Gallery Problem is NP hard [DVOS00]. Therefore to find the optimal solution for a particular Art Gallery Problem, we need an algorithm that generates all triangulations of a given simple polygon.

1.3 Challenges

In this section we discuss the main challenges that any algorithm for enumerating combinatorial objects must face [Sav97]. We have considered all these challenges while developing our algorithms in this thesis and have given algorithmic techniques that successfully resolve the difficulties mentioned in the following subsections.

1.3.1 Time Complexity

The number of different objects is very large in many cases. For example, the number of different permutations of n numbers is exponential. Therefore, to generate all the objects of a particular combinatorial class, we may have to find an exponential number of objects. That means, the overall time complexity of the algorithm is at best exponential, which means the generation of individual objects must be very efficient. There are a number of techniques that accomplish the task. We mention some of those techniques in Section 2.5.

1.3.2 Avoiding Duplications

In any enumeration algorithm, we must have a way to avoid generation of redundant objects. One way to avoid duplications of objects is to store each object generated so far and check each newly generated object with all the previous one to find whether the newly generated one is a duplication. This way of checking duplications has two problems. First, the time complexity goes up. Second, the space requirement becomes very high. We mention some alternatives for avoiding duplications in Section 2.5.

1.3.3 I/O Operations

Algorithms that solve enumeration problems are generally I/O intensive and the output of the algorithm dominates the running time. This is because the number of objects generated is exponential in many cases and each of these objects must be output to an output device. Since I/O is slower than computation, the more I/O operations an algorithm performs the slower it becomes. For this reason reducing the amount of output is essential

1.3.4 Exhaustive Generation

While we exhaustively generate combinatorial objects, we must have an efficient way to determine the end of generation. One solution to this problem is that we count the number of objects generated so far and check whether we have explored all the possibilities. But this works only in the case where we know in advance the total number of distinct objects to be generated and have an efficient way for detecting repetitions. For many problems, it may be difficult to know or calculate the exact number objects that will be generated. For example, it is not trivial to count the number of different triangulations of a given arbitrary plane graph.

1.4 Goals of an Enumeration Algorithm

Any algorithm for generating all objects of a particular combinatorial class has to achieve a number of goals or aims. We list the most important ones below.

- Reduce the time complexity,
- Minimize the usage of memory,
- Reduce the amount of output,
- Avoid duplications, and
- Avoid omissions.

In this thesis, we have considered each of these goals while we develop our algorithms. To achieve the goals, we have developed efficient representations of objects, efficient data structure for storage, and clever algorithmic techniques. We will address the issues mentioned above while we describe our algorithms in detail in later chapters.

1.5 Literature Review

Triangulations play a central role in Computational Geometry and there is a growing body of papers considering triangulations of a point set [Aic99, ES94, HNU99, HOS96]. There are also some well known results for triangulating simple polygons [DVOS00, Cha91] and finding bounds on the number of operations required to transform one triangulation into another [KNN99, STT88].

Let S be a set of n points in general position in the plane. Bespamyatnikh [Bes02] gave an algorithm that generates all the triangulations of S in $O(\log \log n)$ time per triangulation. Avis and Fukuda [AF96] devised a reverse search method which allows to enumerate triangulations in $\theta(nt(S))$ time, where $t(S)$ is the number of triangulations of S. It uses the well known process of the construction of the Delaunay triangulation [For87]. Researchers also have focused their attention on generating triangulated polygons and graphs with certain properties [Avi96, Nak02, NU04b]. The basic operation on triangulations in a $flip$, which can be defined as follows. When two adjacent triangles form a convex quadrilateral then the shared diagonal can be *flipped* and a new triangulation of S is obtained. The graph $G(S)$, called the *graph of triangulations* of a given polygon or point set S, is the graph where the vertices of $G(S)$ are the triangulations of S and two triangulations being adjacent if one can be obtained from the other by flipping an edge. These graphs are widely studied in [HNU99]. A large number of researchers have studied the graph of triangulations $G_T(n)$ of convex polygons of n vertices [HN99, Lee89, STT88]. Figure 1.7 shows the graph of triangulations of convex polygons of six vertices.

In the following two subsections, we review two algorithms that are more closely related to the problem we have addressed in this thesis. We also discuss the limitations of both algorithms.

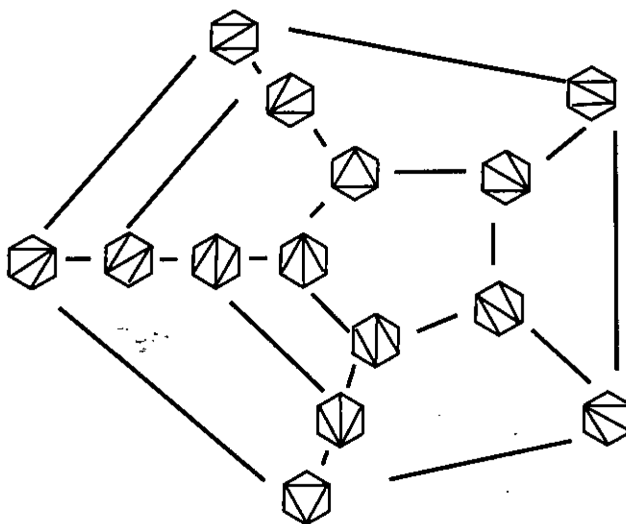


Figure 1.7: Graph of triangulations of convex polygons of six vertices.

1.5.1 Algorithm of Hurtado and Noy

Hurtado and Noy [HN99] built a tree of triangulations of convex polygons with any number of vertices. Figure 1.8 shows the first four levels of the tree. Their construction is primarily of theoretical interests; also all the triangulations of convex polygons with number of vertices less than n need to be found before finding the triangulations of a convex polygon of n vertices. That is, the algorithm of Hurtado and Noy starts with smallest possible triangulated convex polygon, which is a triangle. From it, all the triangulations of a convex polygon of four vertices are generated. From these triangulations, all the triangulations of a convex polygon of five vertices are generated and so on. Figure 1.9 and 1.10 illustrate the operations used to generate new triangulations from old ones. The idea is to split the vertex v_n into two vertices, v_n neighboring v_{n-1} and v_{n+1} neighboring v_1. The same splitting occurs also to the diagonals incident to v_n.

Therefore,if all we want is the triangulations of a convex polygon of n vertices, then the algorithm of Hurtado and Noy is not the desired algorithm, since in this case all the triangulations of convex polygons of number of vertices less than n will be generated also. In this thesis, our algorithm for generating all triangulations of a convex polygon of n

vertices does not suffer from this problem.

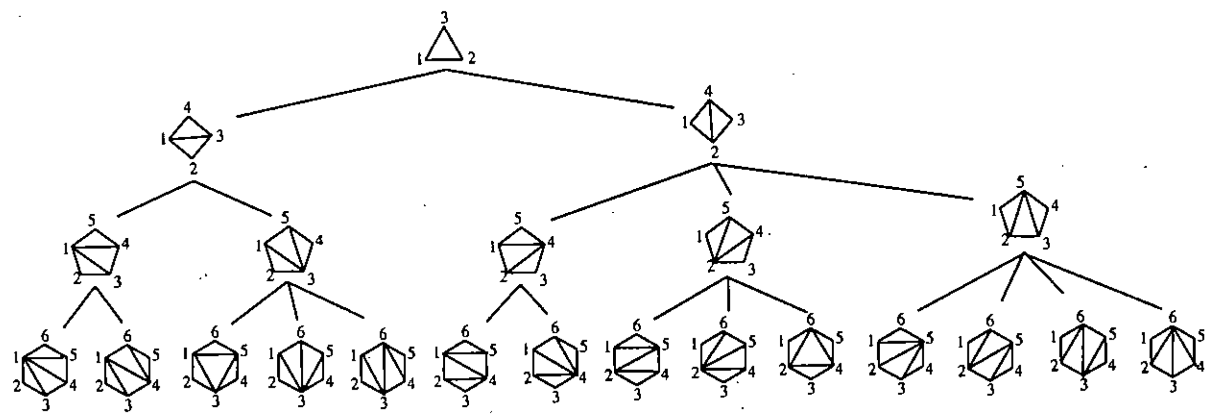


Figure 1.8: Illustration of tree of triangulations of convex polygons.

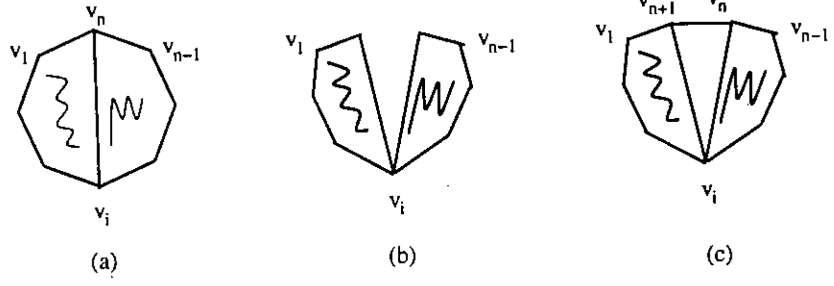


Figure 1.9: Illustration of splitting operation.

1.5.2 Algorithm of Li and Nakano

Li and Nakano [LN01] gave an algorithm to generate all biconnected "based" plane triangulations with at most n vertices. Figure 1.11 shows the tree of triangulations built by the algorithm of Li and Nakano. Their idea was to generate all graphs with some properties without duplications. Here also, the biconnected "based" plane triangulations of n vertices are generated after the biconnected based plane triangulations of less than n vertices are generated. Hence, if we need to generate the triangulations of a convex polygon or a plane graph of exactly n vertices, existing algorithms generate all the triangulations of convex polygons or plane graphs with less than n vertices. This is not an efficient way of

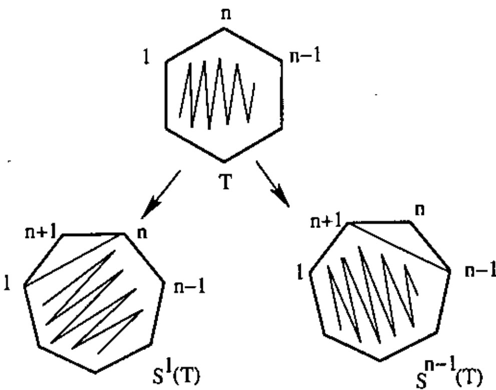


Figure 1.10: Illustration of the generation of new triangulations from an old one.

generation. Also the algorithm of Li and Nakano does not take any graph as input. In this thesis, our algorithm for generating all triangulations of a biconnected plane graph avoids generating such unnecessary triangulations.

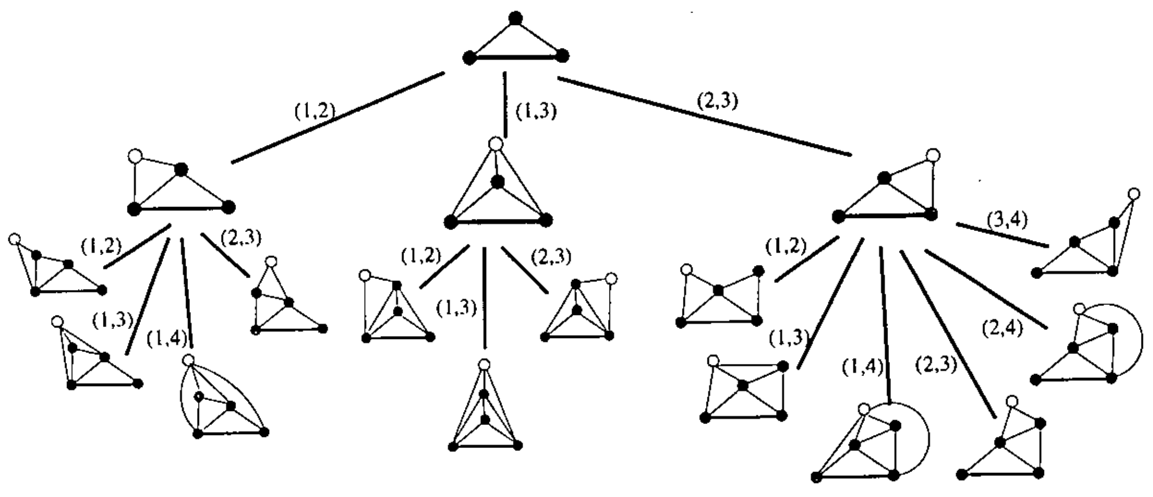


Figure 1.11: Illustration of the tree of triangulations of biconnected based plane triangulations.

1.6 Scope of this Thesis

In this section we list the algorithms we have developed in this thesis.

1.6.1 Labeled Triangulations of a Convex Polygon

The first problem that we consider is finding all the triangulations of a convex polygon P of n vertices, where the vertices of P are numbered. The triangulations of P in this case are called *labeled triangulations* . We give an algorithm that generates all the labeled triangulations of a convex polygon P of n vertices. The algorithm generates each triangulations in $O(1)$ time per triangulation and uses $O(n)$ space. We give the detailed algorithm in Chapter 3.

1.6.2 Triangulations of a Biconnected Plane Graph

The second problem that we consider in this thesis is to generates all the triangulations of a given plane graph G. Specifically we consider the cases where the graph G is either a biconnected outerplanar graph or a biconnected plane graph. For each of these cases, we give algorithm that generates all the triangulations of the given input graph G in time $O(1)$ per triangulation using $O(n)$ space. We give the detailed algorithm in Chapter 4.

1.6.3 Unlabeled Triangulations of a Convex Polygon

Finally we consider the problem of generating all the triangulations of a convex polygon P of n vertice, where the vertces of P are not numbered. The triangulations of P in this case are called *unlabeled triangulations* . We give an algorithm that generates each unlabeled triangulation of P in $O(n^2)$ time worts case time using $O(n)$ space. We give the detailed algorithm in Chapter 3.

1.7 Summary

In this thesis we give efficient algorithms for generating all triangulations of plane graphs and convex polygons. Our main results can be divided into two parts.

Criteria	Li and Nakano	Hurtado and Noy	Our algorithm
Generates	Biconnected Plane Triangulations	Triangulated Convex Polygons	Triangulations of Plane Graphs and Convex Polygons
Takes Input?	NO	NO	YES
Redundant Objects	YES	YES	NO
Generation Time per Object	$O(1)$	$O(1)$	$O(1)$
Space Complexity	$O(n)$	$O(n)$	$O(n)$
References	[LN01]	[HN99]	Ours

Table 1.1: Comparison Table.

The first part of the results is about the triangulations of convex polygons. We give an efficient algorithm that generates all triangulations of a convex polygon P of n vertices where the vertices of P are numbered. The algorithm generates each triangulation of P in constant time from previous one and uses linear space. We also give an algorithm for generating all triangulations of a convex polygon P of n vertices where the vertices of P are not numbered. The algorithm generates each triangulation of P in $O(n^2)$ time per triangulation from previous one and uses linear space.

The second part of the results deals with the plane graphs. Using the algorithm for generating all triangulations of a convex polygon P of n vertices, we give algorithms that generate all triangulations of a biconnected outerplanar graph and biconnected plane graph. The algorithms generate each triangulation in constant time from its previous one using linear space only.

A comparison between our algorithms and the algorithms of Li and Nakano [LN01] and Hurtado and Noy [HN99] is made in Table 1.1 for number of criteria.

Chapter 2

Preliminaries

In this chapter we define some basic terms of graph theory and algorithms. Definitions which are not included in this chapter will be introduced as they are needed. We start, in Section 2.1, by giving definitions of some standard graph-theoretical terms used throughout the remainder of this thesis. We describe some notions from complexity theory in Section 2.2. Sections 2.3 and 2.4 deal with the graph traversal and face traversal algorithms, respectively. Section 2.5 deals with the well known techniques for solving enumeration problems. Finally, Section 2.6 deals with the Catalan Families of combinatorial objects.

2.1 Basic Terminology

In this section we give definitions of some graph-theoretical terms used throughout the remainder of this thesis. Readers interested in graph theory may consult [Wes01].

2.1.1 Polygon

A *polygon* is the region of a plane bounded by a finite collection of line segments forming a simple closed curve. Each line segment of the closed curve is called a *side* or an *edge* of the polygon. A point joining two consecutive sides of a polygon is called a *vertex* of the

polygon. A polygon is called *simple* if it does not cross itself. The set of points in the plane enclosed by a simple polygon forms the *interior* of the polygon, the set of points on the polygon itself forms its *boundary*, and the set of points surrounding the polygon forms its *exterior* . We say two vertices x and y of polygon P is *visible* to each other if and only if the *closed line segment* xy is nowhere exterior to the polygon P; i.e, $xy \subseteq P$. We say x has clear visibility to y if $xy \subseteq P$ and xy does not touch any vertex or edge of P. A *diagonal* of a polygon P is a line segment between two of its vertices x and y that are clearly visible to each other.

A simple polygon is *convex* if, given any two points on its boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's boundary or interior. Let the vertices of a convex polygon P are labeled $v_1, v_2, \cdots, v_n$ counterclockwise. We represent P by listing its vertices as $P = \langle v_1, v_2, \cdots, v_n \rangle$, and represent the edges of P by $(v_1, v_2), (v_2, v_3), \cdots, (v_n, v_1)$.

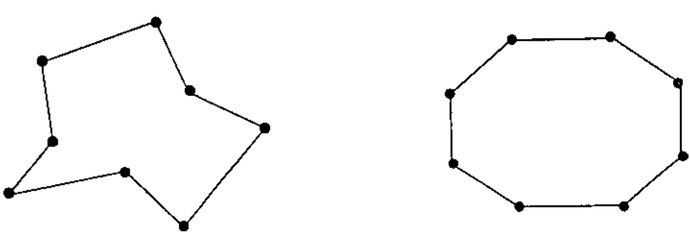


Figure 2.1: Illustration of a polygon.

2.1.2 Graphs

A *graph* G is a structure (V, E) which consists of a finite set of *vertices* V and a finite set of *edges* E; each edge is an unordered pair of distinct vertices. We denote the set of vertices of G by $V(G)$ and the set of edges by $E(G)$. Figure 2.2 illustrates an example of a graph. An edge connecting vertices v_i and v_j in V is denoted by (v_i, v_j). An edge (v_i, v_j) is called a *loop* if $v_i = v_j$. A graph is called a *simple graph* if there is no loop or multiple edges between any two vertices in G.

Let $G = (V, E)$ be a undirected connected simple graph with vertex set V and edge set E. In this thesis, to make the data structures easier to manipulate, we write the edge (v_i, v_j) such that $i < j$. Thus the edge incident to vertex v_4 and v_1 is denoted by (v_1, v_4), and not by (v_4, v_1).

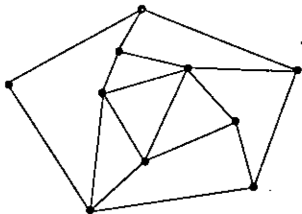


Figure 2.2: Illustration of a graph.

The *degree* of a vertex v is the number of edges incident to v in G. The *connectivity* $\kappa(G)$ of a graph G is the minimum number of vertices whose removal results in a disconnected graph or a single vertex graph. A graph is *k-connected* if $\kappa(G) \geq k$.

2.1.3 Planar graph and plane graph

A graph is *planar* if it can be embedded in the plane so that no two edges intersect geometrically except at a vertex to which they are both incident. Note that a planar graph may have an exponential number of embeddings. Figure 2.3 shows four planar embeddings of the same planar graph.

A *plane graph* is a planar graph with a fixed embedding. A plane graph divides the plane into connected regions called *faces* . The unbounded face is called *outer face* and the other faces are called *inner faces* . For example, the plane graph in Figure 2.3(a) has five inner faces and one outer face.

2.1.4 Triangulations of a Polygon

Let $P = \langle v_1, v_2, \cdots, v_n \rangle$ is a simple polygon. A diagonal (v_i, v_j) divides the polygon P into two polygons: $\langle v_i, v_{i+1}, \cdots, v_j \rangle$ and $\langle v_j, v_{j+1}, \cdots, v_i \rangle$. A decomposition of a polygon

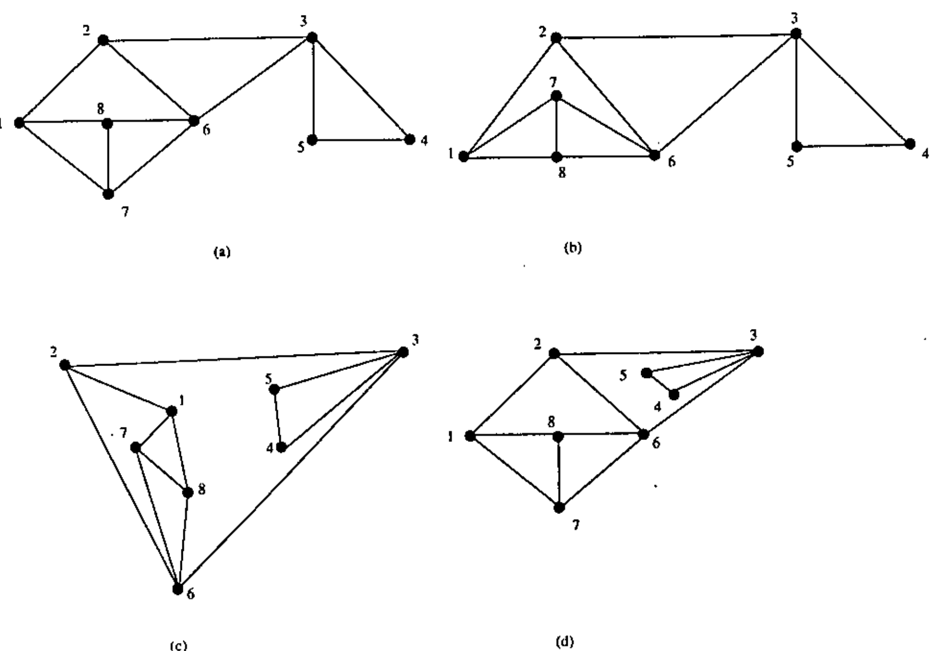


Figure 2.3: Illustration of planar embedding.

into triangles by a set of non-intersecting diagonals is called a *triangulation* of the polygon. Figure 2.4(a) illustrates a triangulated simple polygon, whereas Figure 2.4(b) is an example of a triangulated convex polygon.

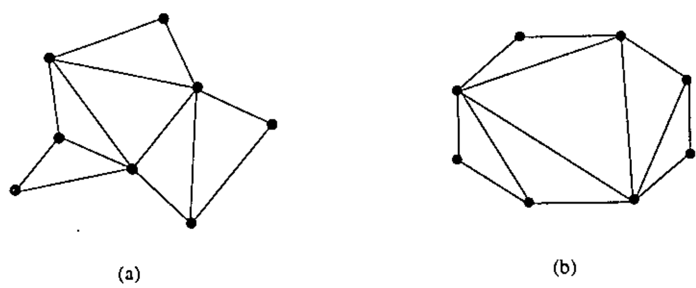


Figure 2.4: Illustration of triangulated polygons (a) triangulation of a simple polygon and (b) triangulation of a convex polygon.

A simple polygon may have many different triangulations. Figure 2.5 shows two different triangulations of a convex polygon P. The set of diagonals is maximal in a triangulation T; that means, every diagonal not in T intersects some diagonal in T. The

sides of triangles in the triangulation are either the diagonals or the sides of the polygon. Every triangulation of a convex polygon P of n vertices has $n-3$ diagonals and $n-2$ triangles.

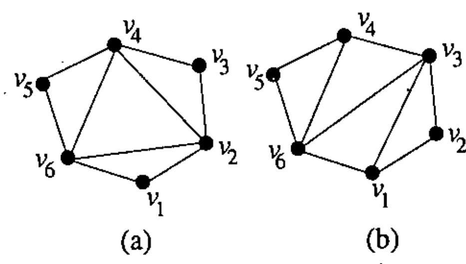


Figure 2.5: Two ways of triangulating a convex polygon of 6 vertices.

Throughout this thesis, we represent each triangulation T of a convex polygon P by listing its diagonals. For example, the triangulation of Figure 2.5(a) is represented by $T = \{(v_4, v_6), (v_2, v_6), (v_2, v_4)\}$. Given the list of diagonals, we can uniquely construct the corresponding triangulation.

2.1.5 Labeled and Unlabeled Triangulations

A triangulation T of a simple polygon P of n vertices is called a *labeled triangulation* if the vertices of P are numbered sequentially from v_1 to v_n. Both triangulations of Figure 2.6 are labeled triangulations. On the other hand, if the vertices of P are not numbered, then the triangulations of P are called *unlabeled triangulations* . Both triangulations of Figure 2.7 are examples of unlabeled triangulations.

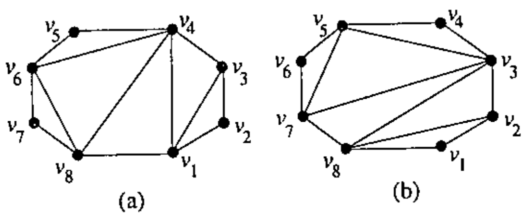


Figure 2.6: Illustration of labeled triangulations.

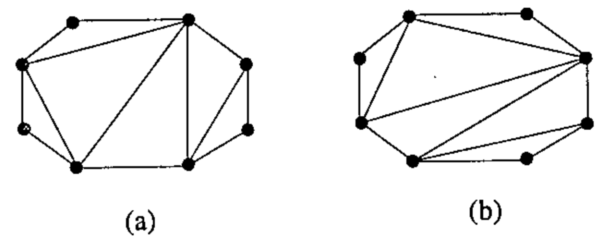


Figure 2.7: Illustration of unlabeled triangulations.

2.1.6 Triangulations of a Graph

The triangulations of a plane graph can be defined analogously. Consider the plane graph G shown in Figure 2.8(a). G has five faces. Consider the face $F = < v_1, v_4, v_5, v_6 >$. If we add an edge between v_1 and v_5, as illustrated in Figure 2.8(b), then the face F is divided into two smaller faces F_1 and F_2. Both F_1 and F_2 have three edges on their boundary. We say that by adding the edge (v_1, v_5) in G we have triangulated the face F of G. By adding additional edges in G, we can triangulate the other faces of G. When all the faces of G are triangulated, we get a triangulation of the graph G, as shown in Figure 2.8(c).

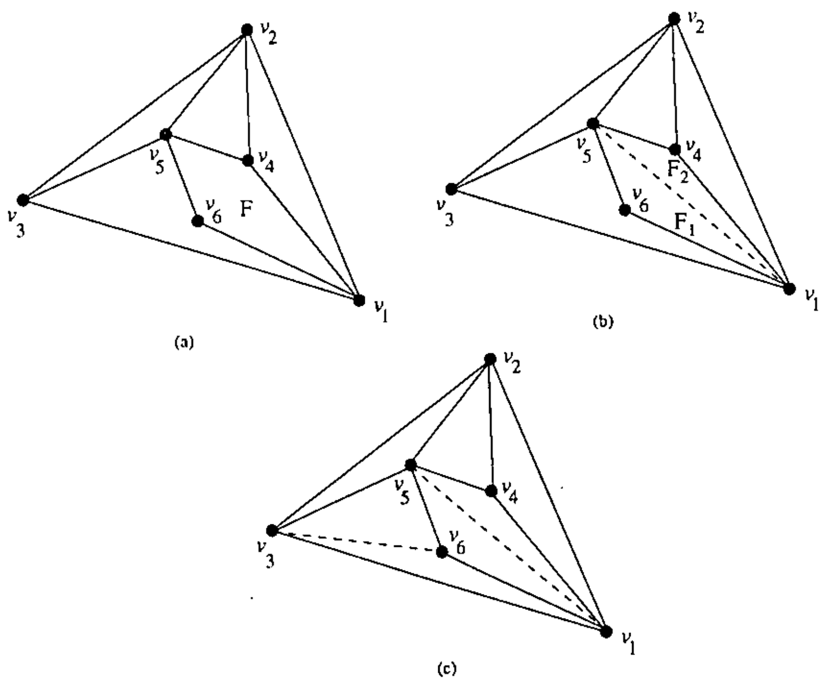


Figure 2.8: Illustration of a triangulated graph.

2.1.7 Graphs and Polygons

There is a plane graph G naturally associated with a triangulation T of a convex polygon P. The vertices of G are the vertices of the triangulation and the edges of G are the sides and diagonals of the triangulation. When there is no confusion, we will treat the triangulation T and the plane graph G associated with it as essentially same. We will sometimes use the terminology of graphs while discussing the triangulations of polygons. Thus, we say that the vertices v_i and v_j of a convex polygon P are adjacent, if (v_i, v_j) is a diagonal or a side of P and call (v_i, v_j) an edge incident to vertex v_i and v_j.

2.1.8 Paths and Cycles

A $v_0 - v_l$ *walk* , $v_0, e_1, v_1, \cdots, v_{l-1}, e_l, v_l$, in G is an alternating sequence of vertices and edges of G, beginning and ending with a vertex, in which each edge is incident to two vertices immediately preceding and following it. If the vertices $v_0, v_1, \cdots, v_l$ are distinct (except possibly v_0, v_l), then the walk is called a *path* and usually denoted either by the sequence of vertices $v_0, v_1, \cdots, v_l$ or by the sequence of edges $e_1, e_2, \cdots, e_l$. The length of the path is l, one less than the number of vertices on the path. A path or walk is closed if $v_0 = v_l$. A closed path containing at least one edge is called a *cycle* .

2.1.9 Outerplanar Graph

A graph is *outerplanar* if it has an embedding with every vertex on the boundary of the outer face. If the outerplanar graph is biconnected then all the vertices of the graph are on a cycle. Figure 2.9 shows examples of outerplanar and biconnected outerplanar graphs.

2.1.10 Trees

A *tree* is a connected graph containing no cycle. Figure 2.10 is an example of a tree. The vertices in a tree are usually called *nodes* . A *rooted tree* is a tree in which one of the

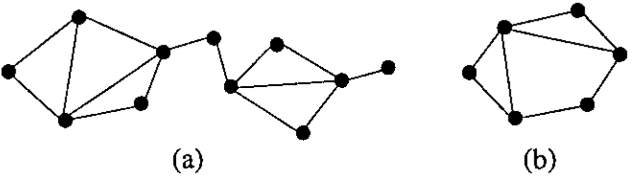


Figure 2.9: Example of (a) an outerplanar graph and (b) biconnected outerplanar graph.

nodes is distinguished from the others. The distinguished node is called the *root* of the tree. The root of a tree is generally drawn at the top. In Figure 2.10, the root is v_1. Every node u other than the root is connected by an edge to some other node p called the *parent* of u. We also call u a *child* of p. We draw the parent of a node above that node. For example, in Figure 2.10, v_1 is the parent of v_2, v_3 and v_4, while v_2 is the parent of v_5 and v_6; v_2, v_3 and v_4 are children of v_1, while v_5 and v_6 are children of v_2. A *leaf* is a node of a tree that has no children. An *internal node* is a node that has one or more children. Thus every node of a tree is either a leaf or an internal node. In Figure 2.10, the leaves are v_4, v_5, v_6, v_7 and v_8, and the nodes v_1, v_2 and v_3 are internal nodes.

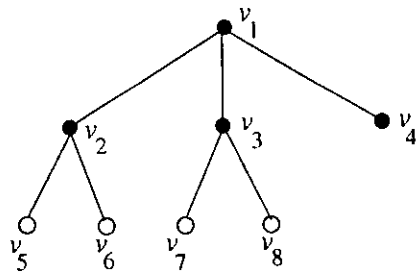


Figure 2.10: Illustration of a tree.

The parent-child relationship can be extended naturally to ancestors and descendants. Suppose that $u_1, u_2, \cdots, u_l$ is a sequence of nodes in a tree such that u_1 is the parent of u_2, which is a parent of u_3, and so on. Then node u_1 is called an *ancestor* of u_l and node u_l a *descendant* of u_1. The root is an ancestor of every node in a tree and every node is a descendant of the root. In Figure 2.10, all seven nodes are descendants of v_1, and v_1 is an ancestor of all nodes.

The *height* of a node u in a tree is the length of a longest path from u to a leaf. The height of the tree is the height of the root. The *depth* of a node u in a tree is the length of a path from the root to u. The *level* of a node u in a tree is the height of the tree minus the depth of u. In Figure 2.10, for example, node v_2 is of height 1, depth 1 and level 1. The tree in Figure 2.10 has height 2.

2.1.11 Binary Trees

A *binary tree* is either a single node or consists of a node and two subtrees rooted at the node, both of the subtrees are binary trees. Figure 2.11 illustrates a binary tree of 15 nodes.

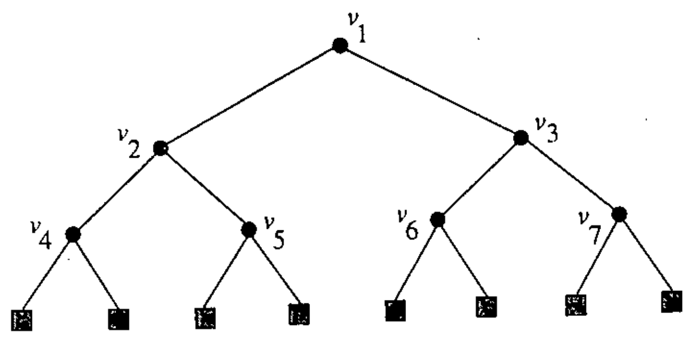


Figure 2.11: Illustration of a binary tree.

2.1.12 Degree Sequence

Let G be a plane graph of n vertices where the vertices are labeled v_1 to v_n. The *degree sequence* of G is the sequence $\langle d_1, d_2, \cdots, d_n \rangle$, where d_i is the degree of the vertex v_i. For example, the degree sequence for the graph of Figure 2.8(a) is $\langle 4, 4, 3, 3, 4, 2 \rangle$

2.2 Algorithms and Complexity

In this section we briefly introduce some terminologies related to complexity of algorithms.

The most widely accepted complexity measure for an algorithm is the *running time* which is expressed by the number of operations it performs before producing the final answer. The number of operations required by an algorithm is not the same for all problem instances. Thus, we consider all inputs of a given size together, and we define the complexity of the algorithm for that input size to be the worst case behavior of the algorithm on any of these inputs. Then the running time is a function of size n of the input.

2.2.1 The notation $O(n)$

In analyzing the complexity of an algorithm, we are often interested only in the "asymptotic behavior," that is, the behavior of the algorithm when applied to very large inputs. To deal with such a property of functions we shall use the following notations for asymptotic running time. Let $f(n)$ and $g(n)$ are the functions from the positive integers to the positive reals, then we write $f(n) = O(g(n))$ if there exists positive constants c_1 and c_2 such that $f(n) \leq c_1 g(n) + c_2$ for all n. Thus the running time of an algorithm may be bounded from above by phrasing like "takes time $O(n^2)$."

2.2.2 Polynomial algorithms

An algorithm is said to be *polynomially bounded* (or simply *polynomial*) if its complexity is bounded by a polynomial of the size of a problem instance. Examples of such complexities are $O(n)$, $O(n \log n)$ $O(n^{100})$, etc. The remaining algorithms are usually referred as *exponential* or *non-polynomial*. Example of such complexity are $O(2^n)$, $O(n!)$, etc.

When the running time of an algorithm is bounded by $O(n)$, we call it a *linear-time* algorithm or simply a *linear* algorithm.

2.2.3 Complexity of Graph Algorithms

We measure the complexity of an algorithm as a function of the size of the input to the algorithm. In this thesis, the inputs to our algorithms are plane graphs. The size of an input planar graph is measured by the amount of memory needed to represent the graph in a computer, which in turn is a function of the number of edges of the graph.

Since in this thesis, we deal with plane graphs only. In a planar graph, the number of edges m is less than $3n$, where n is the number of vertices of the graph [Wes01]. Therefore, we analyze the complexity of the algorithms of this thesis as a function of n.

2.3 Graph Traversal Algorithm

When designing algorithms on graphs, we often need a method for exploring the vertices and edges of a graph. In this section we describe such a method named *depth first search* (DFS).

In DFS each edge is traversed exactly once in the forward and reverse directions and each vertex is visited. Thus DFS runs in linear time. We now describe the method.

Consider visiting the vertices of a graph G in the following way. We select and visit a starting vertex v. Then we select any edge (v, w) incident on v and visit w. In general, suppose x is the most recent visited vertex. The search is continued by selecting some unexplored edge (x, y) incident on x. If y has been previously visited, we find another new edge incident on x. If y has not been visited previously, then we visit y and begin a new search starting at y. After completing the search through all paths beginning at y, the search returns to x, the vertex from which y was first reached. The process of selecting unexplored edges incident to x is continued until the list of these edges is exhausted. This method is called *depth-first search* since we continue searching in the deeper direction as long as possible.

If the graph G is a tree, then we can order the vertices based on the way the edges are

chosen to be traversed. Consider a vertex v from which a new edge would be explored and another vertex would be reached. We mark a vertex u when we first reach u and call the label of u the *rank* of u. The rank of the root of the tree is 0. So the rank of a vertex u is the number of vertices explored before u is reached for the first time. Such a traversal is called a *pre-order* traversal of the vertices of the tree. If a vertex u is labeled after all vertices located in the subtree rooted at u are labeled, then the traversal is called *post-order* traversal. In case of a binary tree, if the vertex u is labeled after all vertices located in the left-subtree rooted at u are labeled, but before all vertices located in the right-subtree rooted at u are labeled, then the traversal is called *in-order* traversal.

2.4 Face Traversal

In this section we describe a data structure to represent a plane graph. We also discuss a face traversal algorithm for a plane graph.

2.4.1 Data Structures for a Plane Graph

A graph can be represented in a computer by either using a matrix or an adjacency list representation. But these two representations are not suitable for representing a plane graph; since a plane graph has a fixed embedding in plane and in a plane graph the edges incident to a vertex have some order which is not preserved by these two representations. Therefore, to represent a plane graph G we use a variation of the adjacency list representation where the ordering of the edges incident to a vertex is preserved. For example, the adjacency list in Figure 2.12(b) represents the plane graph in Figure 2.12(a) since the representation preserves the clockwise ordering of the edges incident to each vertex.

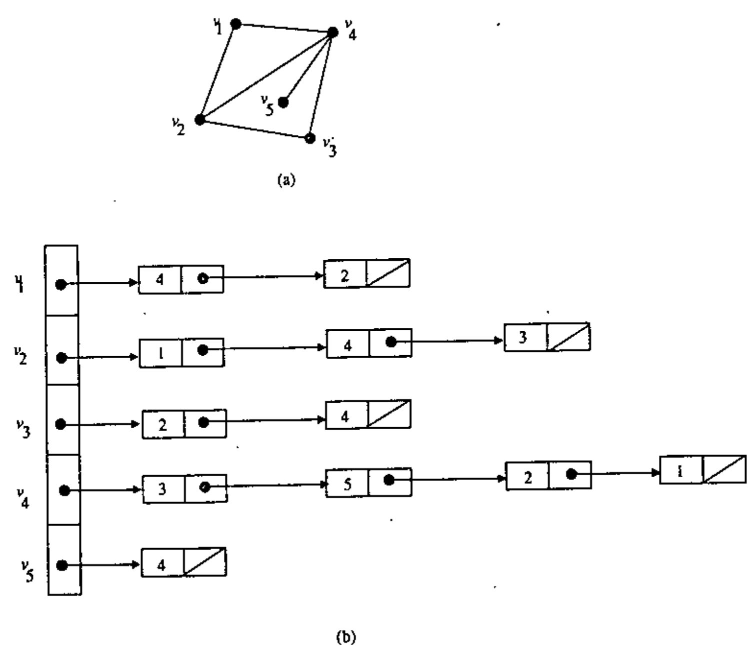


Figure 2.12: Adjacency list representation of a plane graph.

2.4.2 Face Traversal Algorithm

In our algorithm for generating all triangulations of a plane graph G, we need to traverse the faces of G in order to find the root triangulation of the genealogical tree of G. Here we illustrate a special data structure for a plane graph G and describe an algorithm to traverse the faces of G efficiently using the data structure.

Assume we want to clockwise traverse the face F of the graph G starting from vertex v_1 as shown in Figure 2.13(a). To traverse the face efficiently, we represent the graph G using the data structure shown in the Figure 2.13(b).

As illustrated in the Figure 2.13(a), if we want to clockwise traverse the face F starting from vertex v_1, we first traverse edge (v_1, v_4) and reach at vertex v_4. We now need to traverse the edge (v_4, v_3). Using the data structure shown in Figure 2.13(b), we can find that edge in constant time. Here is how this can be done.

The edge (v_1, v_4) follows the edge (v_4, v_2) in the clockwise ordering of the edges incident to vertex v_4. In other words, the edge (v_4, v_2) is counterclockwise next to edge

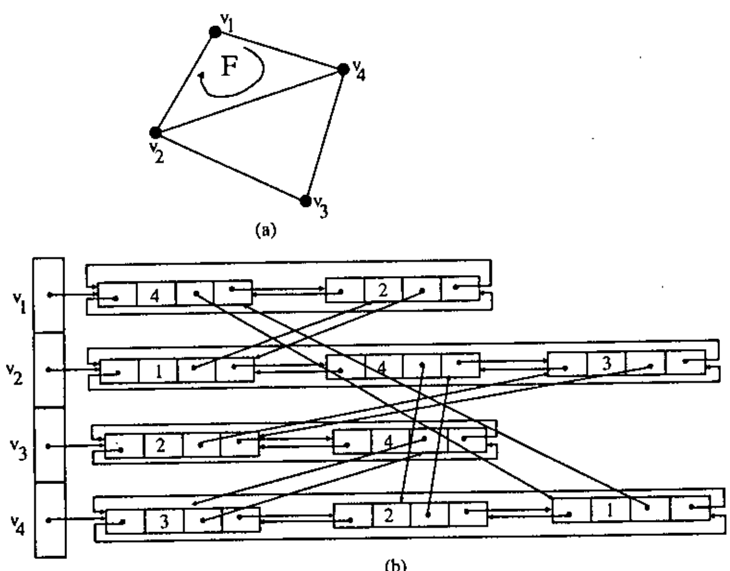


Figure 2.13: Illustration of a data structure for representing a plane graph used for efficient face traversal.

(v_1, v_4) in the adjacent list of v_4. In the representation shown, both the clockwise and counterclockwise ordering of edges incident to a vertex is preserved using a doubly circular linked list of neighbors of the vertices; traversing the list forward and backward we get clockwise and counterclockwise ordering, respectively. The two entries for an edge in the representation are also linked so that one of them can be accessed from the other directly. Using the data structure in Fig. 2.13(b), we can find the edge (v_4, v_2) as follows. From entry 4 in the adjacency list of v_1 go to entry 1 in the adjacency list of v_4 directly using the link between them. Then we can find v_2, since entry 2 is counterclockwise next to entry 1 in adjacency list of v_4.

2.5 Algorithms for Enumeration Problems

There are a number of standard methods that are in use for solving enumeration problems. As mentioned in Chapter 1, there are some difficulties that any enumeration algorithm must resolve somehow. These challenges include reducing the amount of output, efficient

checking for duplications and omissions, space complexity etc. Different methods have different ways of dealing with these challenges.

Classical method algorithms first generate combinatorial objects allowing duplications, but output only if the object has not been output yet. These methods require huge space to store the list of objects generated so far. Furthermore, checking whether the newly generated object will be output takes a lot of time.

Orderly methods algorithms [Mck98] need not to store the list of objects generated so far, they output a object only if it is a canonical representation of an isomorphism class.

Reverse search method algorithms [AF96] also need not to store the list. The idea is to implicitly define a connected graph H such that the vertices of H correspond to the graphs with the given property, and the edges of H correspond to some relation between the graphs. By traversing an implicitly defined spanning tree of H, one can find all the vertices of H, which correspond to all the graphs with the given property.

In the following two subsections, we describe in more detail two other methods for solving enumeration problems and address the techniques employed by these methods for resolving the challenges mentioned above.

2.5.1 Combinatorial Gray Code Approach

To generate all the objects of a particular class, one approach is to try to generate the objects as a list in which successive elements differ only in a small way. The term *Combinatorial Gray Code* first appeared in [JWW80] and is now used to refer to any method for generating combinatorial objects so that successive objects differ in some prespecified, usually small, way. Savage [Sav97] gives a description of the state of the art of the area. The advantages anticipated by such *gray code approach* are manifold. First, generation of successive objects is faster, since each object is generated from the preceding one by making constant number of changes. Secondly, the number of objects in a particular class is generally exponential. Generating algorithms thus produce huge outputs in general,

and the output dominates the running time. If we can reduce the amount of output, the efficiency of the algorithm improves considerably. So in gray code approach, each object is output as a difference from the preceding one, thus removing the necessity to output the entire object. Thirdly, gray codes typically involve elegant recursive constructions which provide new insights into the structure of combinatorial families.

There are many problems that can be solved using combinatorial gray code approach. We list some of them below.

1. Listing all permutations of $1, \cdots, n$,
2. Listing all k-element subsets of an n-element set,
3. Listing all binary trees,
4. Listing all spanning trees of a graph,
5. Listing all partitions of an integer n, and
6. Listing lenear extensions of certain posets etc.

n = 2	n = 4	
1 2	1 2 3 4	4 3 2 1
2 1	1 2 4 3	3 4 2 1
	1 4 2 3	3 2 4 1
	4 1 2 3	2 3 1 4
n = 3	4 1 3 2	2 3 4 1
1 2 3	1 4 3 2	2 4 3 1
1 3 2	1 3 4 2	4 2 3 1
3 1 2	1 3 2 4	4 2 1 3
3 2 1	3 1 4 2	2 4 1 3
2 3 1	3 4 1 2	2 1 4 3
2 1 3	4 3 1 2	2 1 3 4

Figure 2.14: Generating permutations using gray code approach: Johnson-Trotter scheme.

One particular algorithm for generating all permutations of n elements, based on combinatorial gray code approach, is the *Johnson-Trotter algorithm* . Johnson and Trotter

independently showed that it is possible to generate permutations by transpositions even if the two elements exchanged are required to be in adjacent positions. [Tro62, Joh63] The recursive scheme, as shown in Figure 2.14, inserts into each permutation on the list for $n-1$ the element 'n' in each of the possible n positions, moving alternately from right to left, then from left to right.

2.5.2 Family Tree Approach

In the *family tree* or *genealogical tree* approach, a hierarchical structure or tree structure is established among the members of a particular combinatorial class. The idea is to find a unique parent-child relationship among the objects such that one object can be generated from its parent by making a minimal amount of changes. The main feature of this approach is that the entire list of objects need not to be in the memory at once for checking duplications. The objects are generated in the order they are present in the family tree and generation rule itself ensures that no omissions occur. The space complexity for this approach is also linear in the size of an individual object. The main challenge in solving an enumeration problem by family tree approach is to establish a unique parent-child relationship among the objects of interest. For many problems, finding a suitable parent-child relationship may be extremely difficult.

There are a number of problems that have been solved by the family tree approach [Bes02, Nak02, NU04a]. Figure 1.11 illustrates the family tree developed by Li and Nakano for their algorithm for generating all based plane triangulations of graphs.

2.6 Catalan Families

In several families of combinatorial objects, the size of the class is bounded by the Catalan Numbers, defined for $n \geq 0$ by

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

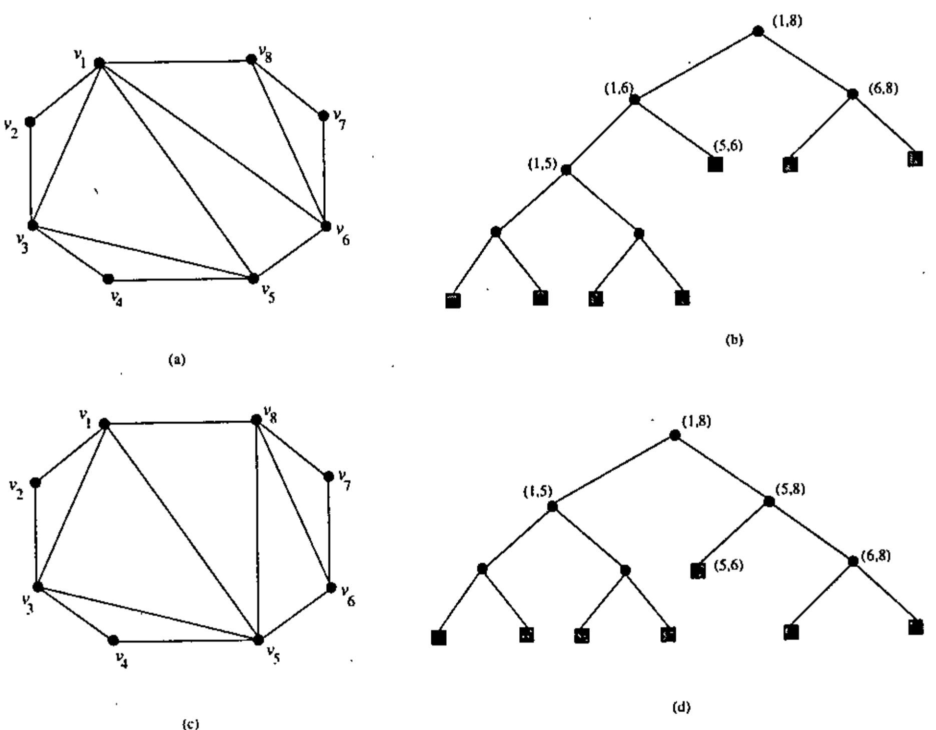


Figure 2.15: Illustration of relationship between triangulations and binary trees.

These include binary trees on n vertices, well formed sequence of $2n$ parentheses, and triangulations of a labeled convex polygon with $n+2$ vertices. There exist bijections between the members of the Catalan family [CLR90]. Therefore, enumeration algorithm for one member of the family gives implicitly a listing scheme for every other member of the family. In Figure 2.15 we show the one to one corresponding between the triangulations of a convex polygon of n vertices and binary trees with $n-2$ internal nodes. As shown in Figure 2.15(c) and Figure 2.15(d), a diagonal flip in a triangulation of the convex polygon is directly related to a rotation in the corresponding binary tree.

Chapter 3

Triangulations of Convex Polygons

3.1 Introduction

In this chapter, we give algorithms to generate all triangulations of a convex polygon P of n vertices. We first consider the case where the vertices of P are numbered sequentially and develop an algorithm that generates all labeled triangulations of P without any duplications or omissions. We also deploy schemes that reduces the amount of output and minimizes the I/O operations. By modifying the algorithm for generating all labeled triangulations of a convex polygon P of n vertices, we give an algorithm that generates all unlabeled triangulations of P, that is the triangulations where the vertices of P are not numbered.

Figure 3.1(a) shows two labeled triangulations of a convex polygon of 4 vertices where Figure 3.1(b) shows an unlabeled triangulation of a convex polygon of 4 vertices.

Based on the algorithms developed in this chapter, we give an algorithm that generates all triangulations of a biconnected plane graph G in Chapter 4.

The rest of the chapter is organized as follows. In Section 3.2 we give the basic idea behind our algorithms in this chapter. Section 3.3 deals with the algorithm for generating all triangulations of a convex polygon P where the vertices of P are numbered sequentially.

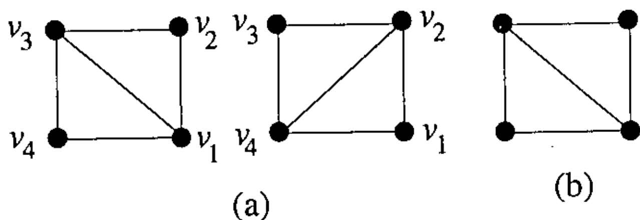


Figure 3.1: Illustration of (a) labeled and (b) unlabeled triangulations.

Section 3.4 gives the algorithm for generating all triangulations of P, where the vertices of P are not numbered.

3.2 Basic Idea

The basic idea behind the algorithm for generating labeled triangulations of a convex polygon P of n vertices is based on the combinatorial gray code and family tree approach. In our algorithm, a new triangulation is generated from an existing one by making a constant number of changes. The main feature of our algorithm is that, we define a tree structure, that is parent-child relationships, among those triangulations. In such a "tree of triangulation", each node corresponds to a triangulation of the convex polygon and each node is generated from its parent in constant time. In our algorithm, we construct the tree structure among the triangulations in such a way that the parent-child relationship is unique, and hence there is no chance of producing duplicate triangulations. Our algorithm also generates the triangulations *in place*, that means, the space complexity is only $O(n)$. Due to the one-to-one relationship between the triangulations of convex polygon of n vertices and binary trees with $n-2$ internal nodes (see Section 2.6), our algorithm readily gives a way to enumerate all binary trees with $n-2$ internal nodes. The algorithm we give that generates all unlabeled triangulations of a convex polygon P of n vertices is based on the algorithm for labeled triangulations of P and generates each new triangulations in worst case time $O(n^2)$ per triangulation.

3.3 Labeled Triangulations of a Convex Polygon

In this section, we give an algorithm to generate all labeled triangulations of a convex polygon P of n vertices. For that purpose we define a unique parent-child relationship among the triangulations of P so that the relationship among the triangulations of P can be represented by a tree with a suitable triangulation as the root. Figure 3.2 shows such a tree of triangulations of a convex polygon of six vertices. Once such a parent-child relationship of P is established, we can generate all the triangulations of P using the relationship. We need not to build or to store the entire tree of triangulations at once, rather we generate each triangulation in the order it appears in the tree structure.

In our algorithm, we use the following operation to generate a new triangulation from an old one. Let T be a triangulation of a convex polygon of n vertices. Let (v_i, v_j) be a shared diagonal of two adjacent triangles of T which form a convex quadrilateral (v_q, v_i, v_r, v_j). If we remove the diagonal (v_i, v_j) from T and add the diagonal (v_q, v_r), we get a new triangulation T'. The above operation is known as *flipping* and has been used by a number of researchers [For87, HNU99, KNN99]. We say that we have *flipped* the edge (v_i, v_j), and denote the new triangulation T' by $T(v_i, v_j)$.

For example, in Figure 3.3(a), the two triangles $\langle v_1, v_2, v_3\rangle$ and $\langle v_1, v_3, v_4\rangle$ form the quadrilateral $\langle v_1, v_2, v_3, v_4\rangle$ and (v_1, v_3) is the shared diagonal. We remove (v_1, v_3) from the triangulation of Figure 3.3(a) and add the diagonal (v_2, v_4) to generate the triangulation of Figure 3.3(b). Thus, we flip the diagonal (v_1, v_3) of the triangulation of Figure 3.3(a) to generate the triangulation of Figure 3.3(b).

One can observe that, each triangulation of the tree of triangulations in Figure 3.2, except the root, is generated from its parent by flipping a single diagonal. Each arrow is labeled in Figure 3.2 to indicate which diagonal has been flipped to generate a particular child. We call a tree of triangulations of a convex polygon P of n vertices a *genealogical tree* of P and denote it by $\mathcal{T}(P)$. Figure 3.2 illustrates $\mathcal{T}(P)$ of a convex polygon P of six

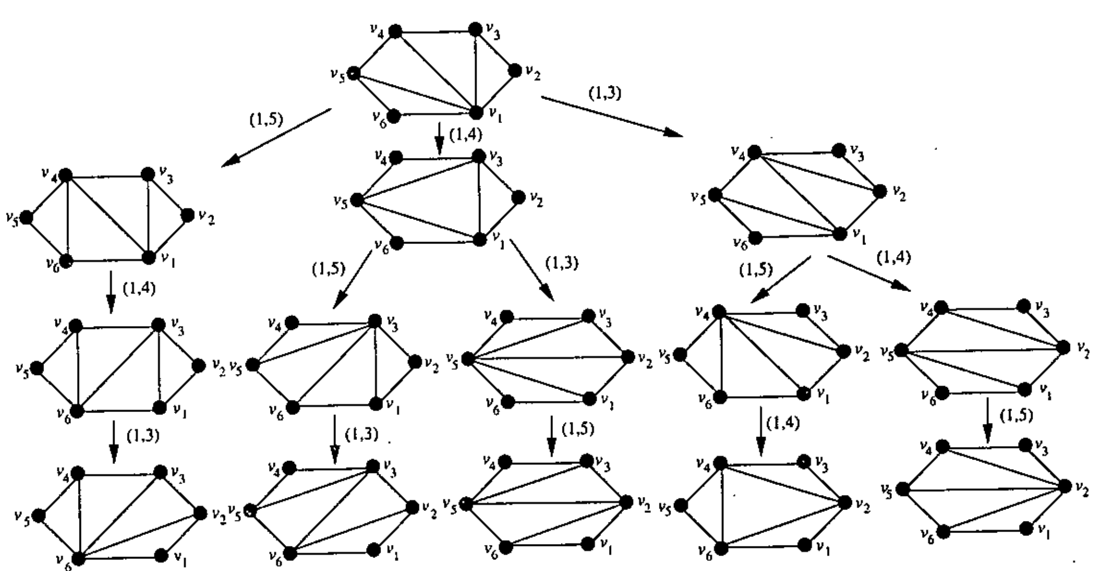


Figure 3.2: Genealogical tree $\mathcal{T}(P)$ for a convex polygon P of six vertices.

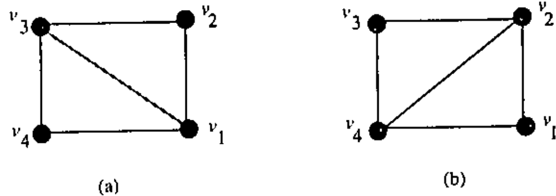


Figure 3.3: Illustration of flipping operation; (a) old triangulation and (b) new triangulation.

vertices. Let T be a triangulation of a convex polygon P of n vertices, in which all the diagonals of T are incident to vertex v_1. We regard T as the *root* T_r of the genealogical tree $\mathcal{T}(P)$. For example, the triangulation in Figure 3.4(b) is the root of the genealogical tree $\mathcal{T}(P)$ of a convex polygon P of 6 vertices as shown in Figure 3.4(a).

Note that, in the root T_r of $\mathcal{T}(P)$, every interior point of P is visible from vertex v_1. We say that vertex v_1 has *full vision* in T_r. Obviously, in a non-root triangulation T of P, vertex v_1 does not have the full vision. The reason is that T has some "blocking diagonals" which are blocking some parts of the convex polygon P from being visible from vertex v_1. A diagonal (v_i, v_j) of a triangulation T of P is a *blocking diagonal* of T if both

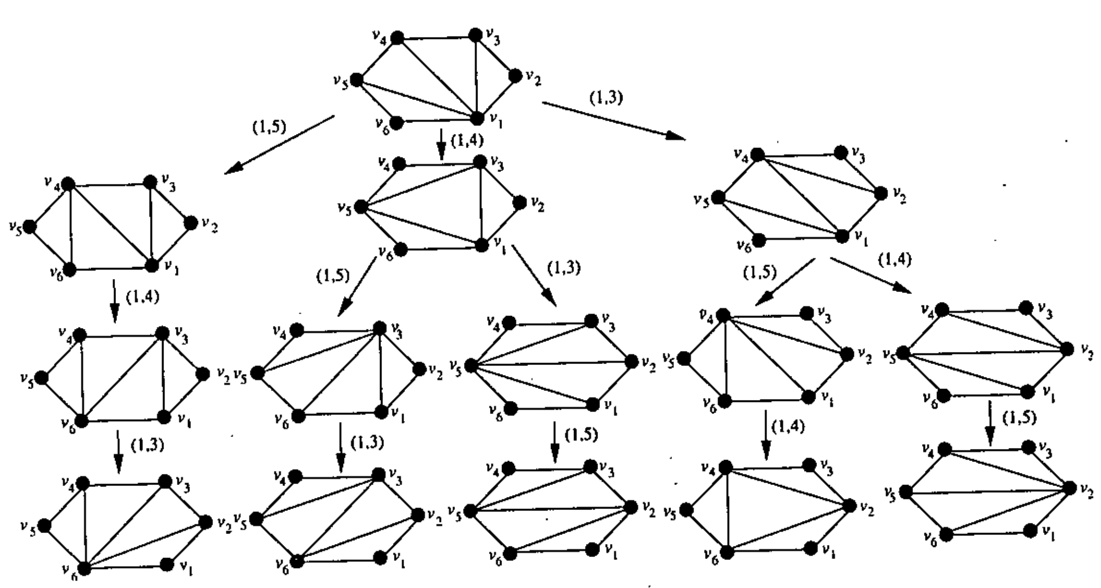


Figure 3.2: Genealogical tree $\mathcal{T}(P)$ for a convex polygon P of six vertices.

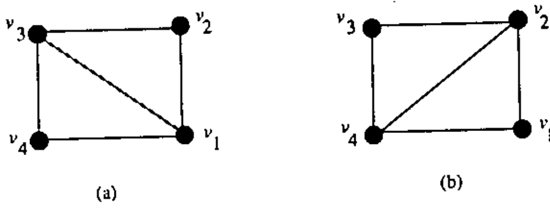


Figure 3.3: Illustration of flipping operation; (a) old triangulation and (b) new triangulation.

vertices. Let T be a triangulation of a convex polygon P of n vertices, in which all the diagonals of T are incident to vertex v_1. We regard T as the *root* T_r of the genealogical tree $\mathcal{T}(P)$. For example, the triangulation in Figure 3.4(b) is the root of the genealogical tree $\mathcal{T}(P)$ of a convex polygon P of 6 vertices as shown in Figure 3.4(a).

Note that, in the root T_r of $\mathcal{T}(P)$, every interior point of P is visible from vertex v_1. We say that vertex v_1 has *full vision* in T_r. Obviously, in a non-root triangulation T of P, vertex v_1 does not have the full vision. The reason is that T has some "blocking diagonals" which are blocking some parts of the convex polygon P from being visible from vertex v_1. A diagonal (v_i, v_j) of a triangulation T of P is a *blocking diagonal* of T if both

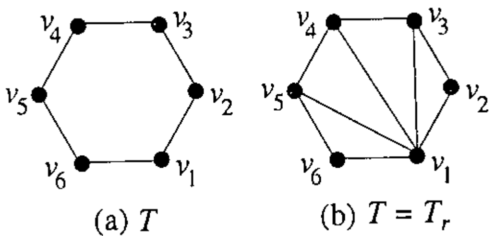


Figure 3.4: Illustration of (a) a convex polygon of six vertices and (b) corresponding root in $\mathcal{T}(P)$.

v_i and v_j are adjacent to v_1 in T. We say that vertex v_1 has *blocked vision* in a non-root triangulation T of P.

The following lemma characterizes the non-root triangulations of a convex polygon P of n vertices.

Lemma 3.3.1 *Each triangulation T of a convex polygon $P = \langle v_1, v_2, \cdots, v_n \rangle$ has at least one blocking diagonal. if T is not the root of $\mathcal{T}(P)$.*

Proof. Let v_j be the vertex of P such that (v_1, v_k) is a diagonal of T, for all $k \geq j$. Then there exists a vertex v_i such that $i < j$ and (v_i, v_j) is a diagonal of T. Otherwise, all diagonals of T would be incident to v_1 and T would be the root of $\mathcal{T}(P)$. Since T is a triangulation of P, (v_1, v_i, v_j) is a triangle, and hence (v_i, v_j) is a blocking diagonal. □

Suppose we flip a diagonal (v_1, v_j) of T to generate a new triangulation T'. Let $(v_b, v_{b'})$, $b < b'$ be the newly found diagonal in T'. Obviously $(v_b, v_{b'})$ is a blocking diagonal of T'. Similarly, if we flip a blocking diagonal of T to generate T', the newly found diagonal will be non-blocking, incident to vertex v_1 in T'. For example, if we flip the diagonal (v_1, v_4) of the triangulation of Figure 3.5(a), we get the triangulation of Figure 3.5(b), where (v_2, v_5) is the newly found diagonal. This new diagonal (v_2, v_5) is a blocking diagonal of the triangulation of Figure 3.5(b).

The rest of this section is organized as follows. Section 3.3.1 describes child to parent relationship among the triangulations of a convex polygon P of n vertices. Section 3.3.2

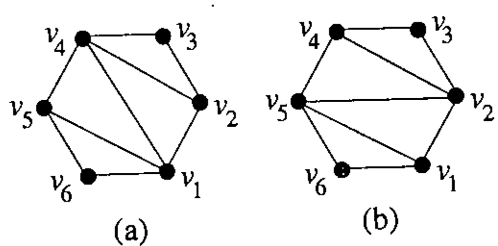


Figure 3.5: Illustration of generation of blocking diagonal; (a) old triangulation and (b) new triangulation.

deals with the generation of children of a triangulation T in the genealogical tree $\mathcal{T}(P)$ of P. Section 3.3.3 describes the data structures used to represent a triangulation T of P. Finally, section 3.3.4 describes the algorithm to generate all the triangulations of P.

3.3.1 Child-Parent Relationship

It is convenient to consider the child-parent relationship before considering the parent-child relationship. Throughout the section, we will denote a triangulation by T and its parent by $P(T)$.

We define the child-parent relationships among the triangulations of P with two goals in mind. First, the differences between a triangulation T and its parent $P(T)$ should be minimal, so that T can be generated from $P(T)$ with minimal effort. Second, every triangulation T of P must have a parent and only one parent in the genealogical tree $\mathcal{T}(P)$. We achieve the first goal by ensuring that the parent $P(T)$ of a triangulation T can be found by flipping a single diagonal of T. That means T can also be found from its parent $P(T)$ by flipping a single diagonal of $P(T)$. The second goal, that is the uniqueness of the parent-child relationship, can be achieved as follows.

Our idea of defining a parent-child relationship is that the parent $P(T)$ of a triangulation T must have a "clearer vision" than T. Let T and T' be two triangulations of P. We say that T' has a *clearer vision* than T if the number of vertices visible from v_1 in T' is more than the number of vertices visible from v_1 in T. For example, three vertices

are visible from vertex v_1 in the triangulation of Figure 3.6(a), whereas four vertices are visible from vertex v_1 in the triangulation of Figure 3.6(b). Therefore the triangulation of Figure 3.6(b) has a clearer vision than the triangulation of Figure 3.6(a). We can easily get a triangulation T' from T, where T' has a clearer vision than T, by flipping a blocking diagonal $(v_b, v_{b'})$ of T. We say that the triangulation T' is the *parent* of T if the diagonal $(v_b, v_{b'})$ is the "leftmost blocking diagonal" of T. The diagonal $(v_b, v_{b'})$, $b < b'$, of T is the *leftmost blocking diagonal* of T if no other blocking diagonals of T is incident to a higher indexed vertex than $v_{b'}$ in T. For example, in the triangulation of Figure 3.6(a), (v_3, v_6) is the leftmost blocking diagonal. Therefore we flip (v_3, v_6) of the triangulation of Figure 3.6(a) to find its parent, which is shown in Figure 3.6(b).

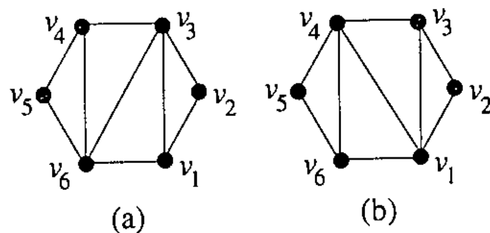


Figure 3.6: Illustration of child-parent relationship; (a) child and (b) parent.

The above definition of the parent of a triangulation T of a convex polygon P ensures that we can always find a unique parent of a non-root triangulation T of P. From Lemma 3.3.1, a non-root triangulation T of P has at least one blocking diagonal, and from those blocking diagonals of T we choose the one which is leftmost and we flip that diagonal to find the unique parent $P(T)$ of T.

Based on the above parent-child relationship, the following lemma claims that every triangulation of a convex polygon P of n vertices in present in the genealogical tree $\mathcal{T}(P)$.

Lemma 3.3.2 *For any triangulation T of a convex polygon $P = \langle v_1, v_2, \cdots, v_n \rangle$, there is a unique sequence of flipping operations that transforms T into the root T_r of $\mathcal{T}(P)$.*

Proof. Let T be a triangulation other than the root of $\mathcal{T}(P)$. Then according to Lemma 3.3.1, T has at least one blocking diagonal. Let $(v_b, v_{b'})$ be the leftmost blocking

diagonal of T. We find the parent $P(T)$ of T by flipping the leftmost blocking diagonal of T. Since flipping a blocking diagonal of T results in a diagonal incident to vertex v_1 in the new triangulation, $P(T)$ has one more diagonals incident to v_1 than T. Now, if $P(T)$ is the root, then we stop. Otherwise, we apply the same procedure to $P(T)$ and find its parent $P(P(T))$. By continuously applying this process of finding the parent, we eventually generate the root triangulation T_r of $\mathcal{T}(P)$. □

Lemma 3.3.2 ensures that there can be no omission of triangulations in the genealogical tree $\mathcal{T}(P)$ of a convex polygon P of n vertices. Since there is a unique sequence of operations that transforms a triangulation T of P into the root T_r of $\mathcal{T}(P)$, by reversing the operations we can generate that particular triangulation, starting at the root. We give the details in the next section.

3.3.2 Generating the Children of a Triangulation in $\mathcal{T}(P)$

In this section we describe the method for generating the children of a triangulation T in $\mathcal{T}(P)$.

To find the parent $P(T)$ of the triangulation T, we flip the leftmost blocking diagonal of T. That means $P(T)$ has fewer blocking diagonals than T. Therefore, the operation for generating the children of T must increase the number of blocking diagonals in the children of T. Intuitively if we flip a diagonal (v_1, v_j) of T, which is incident to vertex v_1 in T, and generate a new triangulation T', then T' contains one more blocking diagonal than T. We call all such diagonals (v_1, v_j) as the *candidate diagonals* of T.

Note that, flipping a candidate diagonal of T may not always preserve the parent-child relationship described in Section 3.3.1. For example, we generate the triangulation of Figure 3.7(b) by flipping the candidate diagonal (v_1, v_3) of the triangulation of Figure 3.7(a). The leftmost blocking diagonal of the triangulation of Figure 3.7(b) is (v_4, v_6); therefore the parent of the triangulation of Figure 3.7(b) is the triangulation of Figure 3.7(c), not the triangulation of Figure 3.7(a).

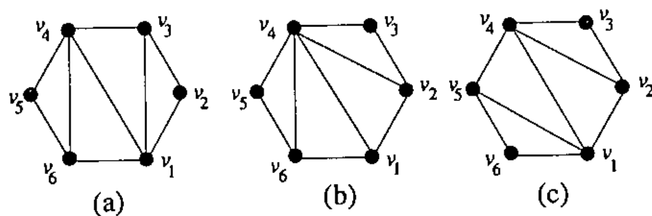


Figure 3.7: Illustration of a flipping operation that does not preserve parent-child relationship.

Therefore to keep the parent-child relationship unique, we flip a candidate diagonal (v_1, v_j) of T to generate a new triangulation T' if only if flipping (v_1, v_j) of T results in the leftmost blocking diagonal of T'. We call such a candidate diagonal (v_1, v_j) of T as a *generating diagonal* . The generating diagonals of a triangulation T of P can be found as follows. Let $(v_b, v_{b'})$ be the leftmost blocking diagonal of a triangulation T of a convex polygon P of n vertices. Then (v_1, v_j) is a generating diagonal of T if $j \geq b$. If T has no blocking diagonal then all diagonals of T are generating diagonals. Thus all the diagonals of the root T_r of $\mathcal{T}(P)$ are generating diagonals. All other candidate diagonals of T are called *non-generating* . We call the set of generating diagonals of a triangulation T as *generating set* C of T. For example, the triangulation in Figure 3.8(a) is the root triangulation of the genealogical tree $\mathcal{T}(P)$ of a convex polygon P of 8 vertices. Therefore, all the diagonals of the triangulation in Figure 3.8(a) are generating diagonals. In the triangulation of Figure 3.8(b), (v_1, v_4), (v_1, v_6) and (v_1, v_7) are three generating diagonals, whereas (v_1, v_3) is a non-generating diagonal.

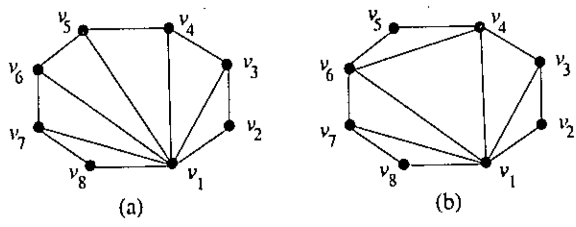


Figure 3.8: Illustration of generating diagonals.

We now have the following lemmas.

Lemma 3.3.3 *The root T_r of the genealogical tree $\mathcal{T}(P)$ of a convex polygon P of n vertices has $n-3$ generating diagonals and any other triangulations in $\mathcal{T}(P)$ has less than $n-3$ generating diagonals.*

Proof. The number of diagonals in any triangulation T of a convex polygon P of n vertices is $n-3$. Thus the maximum number of possible generating diagonals is also $n-3$. Since the root triangulation T_r has all its diagonals as generating, T_r contains $n-3$ generating diagonals. Any triangulation T other than the root T_r contains at least one blocking diagonal, which is not incident to vertex v_1 in T. Since generating diagonals must be incident to vertex v_1, any triangulation other than T_r has less than $n-3$ generating diagonals. □

Lemma 3.3.4 *Let (v_1, v_j) be a generating diagonal of a triangulation T of a convex polygon P of n vertices. Then flipping (v_1, v_j) in T results in the leftmost blocking diagonal of $T(v_1, v_j)$.*

Proof. Let $(v_r, v_{r'})$ be the leftmost blocking diagonal of T. We first consider the case where either $v_j = v_r$ or $v_j = v_{r'}$.

If $v_j = v_r$, then $(v_1, v_j, v_{r'})$ is a triangle of T (see Figure 3.9(a)) and after flipping (v_1, v_j) of T we get $(v_i, v_{r'})$ as a diagonal in $T(v_1, v_j)$, for some $i < j$ (see Figure 3.9(b)). Since every face of $T(v_1, v_j)$ is a triangle, $\langle v_1, v_i, v_{r'}\rangle$ is a triangle of $T(v_1, v_j)$. Therefore, $(v_i, v_{r'})$ is the blocking diagonal of $T(v_1, v_j)$. Since, $(v_r, v_{r'})$ is the leftmost blocking diagonal of T and vertex v_r is not visible from vertex v_1 in $T(v_1, v_j)$, $(v_i, v_{r'})$ is the leftmost blocking diagonal of $T(v_1, v_j)$

If $v_j = v_{r'}$, then $\langle v_1, v_r, v_j\rangle$ is a triangle of T (see Figure 3.9(c)) and after flipping (v_1, v_j) of T we get (v_i, v_r) as a diagonal of $T(v_1, v_j)$, for some $i > j$ (see Figure 3.9(d)). Since every face of $T(v_1, v_j)$ is a triangle, $\langle v_1, v_r, v_i\rangle$ is a triangle of $T(v_1, v_j)$. Therefore, (v_r, v_i) is a blocking diagonal of $T(v_1, v_j)$. Since, $(v_r, v_{r'})$ is a leftmost blocking diagonal

of T and (v_r, v_i) is a blocking diagonal of $T(v_1, v_j)$, where $i > r'$, (v_r, v_i) is the leftmost blocking diagonal of $T(v_1, v_j)$

We now consider the case where $j > r'$ (see Figure 3.9(e)). Let $(v_q, v_{q'})$ be the diagonal which appears in $T(v_1, v_j)$ after flipping the diagonal (v_1, v_j) of T (see Figure 3.9(f)). Every face of $T(v_1, v_j)$ is a triangle. Thus, $\langle v_1, v_q, v_{q'} \rangle$ is a triangle of $T(v_1, v_j)$ and $(v_q, v_{q'})$ is a blocking diagonal of $T(v_1, v_j)$. Since, $q' > j$, we have $q' > r'$. Therefore, $(v_q, v_{q'})$ is the leftmost blocking diagonal of $T(v_1, v_j)$. □

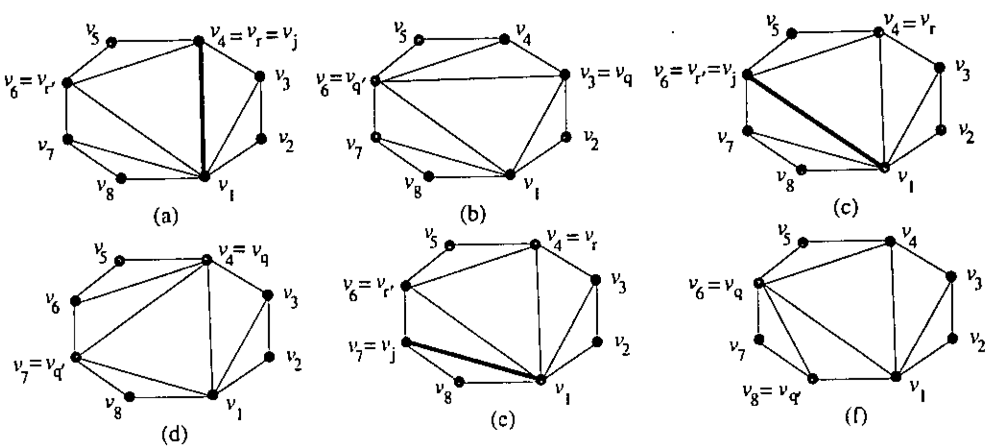


Figure 3.9: Illustration of Lemma 3.3.4.

Lemma 3.3.5 *Let T be a triangulation of a convex polygon P of n vertices. Let $T(v_1, v_j)$ be the triangulation generated by flipping the diagonal (v_1, v_j) of T. Then T is the parent of $T(v_1, v_j)$ in the genealogical tree $\mathcal{T}(P)$ if and only if (v_1, v_j) is a generating diagonal of T.*

Proof. *Necessity.* Assume that (v_1, v_j) is a non-generating diagonal of T. It is sufficient to show that T is not the parent of $T(v_1, v_j)$. Here, we have $j < r$ (see Figure 3.10(a)). Let $(v_q, v_{q'})$ be the diagonal which appears in $T(v_1, v_j)$ after flipping (v_1, v_j) of T (see Figure 3.10(b)). Since the diagonal (v_1, v_r) of T is also a diagonal of $T(v_1, v_j)$, we have $q' \leq r$. Therefore, $q < r'$. Thus, $(v_r, v_{r'})$ is the leftmost blocking diagonal of $T(v_1, v_j)$ and T is not the parent of $T(v_1, v_j)$.

Sufficiency. Assume that (v_1, v_j) is a generating diagonal of T. We show that T is the parent of $T(v_1, v_j)$ in $\mathcal{T}(P)$.

Let $(v_q, v_{q'})$ be the diagonal which appears in $T(v_1, v_j)$ after flipping (v_1, v_j) of T. To prove that T is the parent of $T(v_1, v_j)$ in $\mathcal{T}(P)$, we must show that $(v_q, v_{q'})$ is the leftmost blocking diagonal of $T(v_1, v_j)$.

We first consider the case where T is the root of $\mathcal{T}(P)$. T does not have any parent and all the diagonals of T are incident to vertex v_1. Therefore, $(v_q, v_{q'})$ is the only diagonal of $T(v_1, v_j)$ which is not incident to vertex v_1. Thus, $(v_q, v_{q'})$ is the leftmost blocking diagonal of $T(v_1, v_j)$.

We now consider the case where T is not the root of $\mathcal{T}(P)$. Then by Lemma 3.3.4, $(v_q, v_{q'})$ is the leftmost blocking diagonal of $T(v_1, v_j)$. □

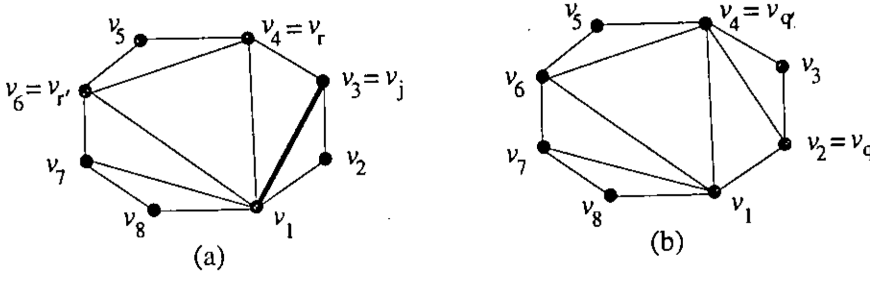


Figure 3.10: Illustration of Lemma 3.3.5.

According to Lemma 3.3.5, if the generating set C of a triangulation T is non-empty, then we can generate each of the children of T in $\mathcal{T}(P)$ by flipping a generating diagonal of T. Therefore, the number of children of a triangulation T in $\mathcal{T}(P)$ will be equal to the cardinality of the generating set. Thus, the following lemma holds.

Lemma 3.3.6 *The number of children of a triangulation T of a convex polygon P is equal to the number of diagonals in the generating set of T. The root of $\mathcal{T}(P)$ has the maximum number of children.*

3.3.3 The Representation of a Triangulation in $\mathcal{T}(P)$

The child generation rule defined in Section 3.3.2 ensures that every triangulation T of a convex polygon P of n vertices is present in the genealogical tree $\mathcal{T}(P)$ and every triangulation T except the root T_r of $\mathcal{T}(P)$ is generated from its parent. In this section we describe a data structure that we use to represent a triangulation T and that enables us to generate each child triangulation of T in constant time.

For a triangulation T of a convex polygon P of n vertices, we maintain three lists: L, C and O to represent T completely. Here L is the list of diagonals of T and C is the generating set of T. For each diagonal (v_1, v_j) in the generating set C of T, we maintain a corresponding *opposite pair* $(v_o, v_{o'})$, such that $\langle v_1, v_o, v_j, v_{o'} \rangle$ is a convex quadrilateral of T. Note that, $o < j$ and $o' > j$. O is the list of list of such opposite pairs. For example, in Figure 3.11, the generating diagonal (v_1, v_4) has the opposite pair (v_3, v_6).

Since we generate triangulations of P starting with the root T_r, we find the representation of T_r first. The diagonals of T_r are listed in L in counterclockwise order. That is, for T_r, $L = \{(v_1, v_{n-1}), (v_1, v_{n-2}), \cdots (v_1, v_3)\}$. The generating set C is exactly similar to the list L of T_r: $C = \{(v_1, v_{n-1}), (v_1, v_{n-2}), \cdots, (v_1, v_4), (v_1, v_3)\}$. Corresponding list of opposite pairs is $O = \{(v_{n-2}, v_n), (v_{n-3}, v_{n-1}), \cdots, (v_3, v_5), (v_2, v_4)\}$; that means, (v_{j-1}, v_{j+1}) is the opposite pair of (v_1, v_j) in T_r, $3 < j < n - 1$.

Let $T(v_1, v_j)$ be a child triangulation of T in $\mathcal{T}(P)$ generated from T by flipping the diagonal (v_1, v_j) of T. Let $(v_b, v_{b'})$ be the blocking diagonal which appears in $T(v_1, v_j)$ after flipping (v_1, v_j) of T. The list L of $T(v_1, v_j)$ can be found easily from the representation of T by removing (v_1, v_j) from the list L of T and adding $(v_b, v_{b'})$ to it. Note that one can easily find the blocking diagonal $(v_b, v_{b'})$ of T', since $(v_b, v_{b'})$ is the opposite pair of (v_1, v_j) in the representation of T.

In the next section we give the detailed algorithm for generating the triangulations of P and show that the representation of a child triangulation T' of T can be found from the representation of T in constant time.

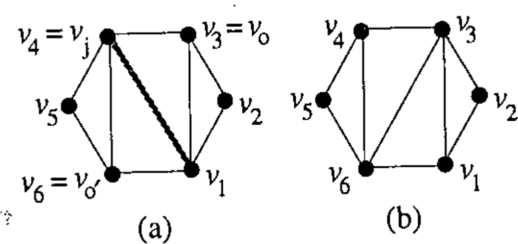


Figure 3.11: Illustration of (a) a generating diagonal with opposite pair and (b) blocking diagonal generated by flipping the generating diagonal.

3.3.4 The Algorithm

In this section we give an algorithm to generate all triangulations of a convex polygon P of n vertices.

Let $v_{j_1}, v_{j_2}, \cdots, v_{j_k}$, $j_1 > j_2 > \cdots > j_k$, be the sequence of k vertices of a triangulation T of P such that $(v_1, v_{j_1}), (v_1, v_{j_2}), \cdots, (v_1, v_{j_k})$ are the diagonals of T and each of the diagonals $(v_1, v_{j_i}), 1 \leq i \leq k$, is a generating diagonal of T. Then, T has a generating set $C = \{(v_1, v_{j_1}), (v_1, v_{j_2}), \cdots, (v_1, v_{j_k})\}$ of k generating diagonals, for $0 \leq k \leq n-3$. For T_r, $C = \{(v_1, v_{n-1}), (v_1, v_{n-2}), \cdots, (v_1, v_4), (v_1, v_3)\}$. For each diagonal (v_1, v_j) of T, we keep an opposite pair $(v_o, v_{o'})$ in T. O is the set of such pairs. For T_r, $O = \{(v_{n-2}, v_n), (v_{n-3}, v_{n-1}), \cdots, (v_3, v_5), (v_2, v_4)\}$ as shown in Section 3.3.3. We find the sets C and O of a child T' of T by updating the lists C and O of T while we generate T'.

We now describe a method for generating the children of a triangulation T in $\mathcal{T}(P)$. We have two cases based on whether T is the root of $\mathcal{T}(P)$ or not.

Case 1: T is the root of $\mathcal{T}(P)$.

In this case, all the diagonals of T are generating diagonals and there are a total of $n-3$ such diagonals in T. Any of these generating diagonals of T can be flipped to generate a child triangulation of T. For example, the root of the genealogical tree in Figure 3.2 has three generating diagonals; thus it has three children as shown in Figure 3.2.

Case 2: T is not the root of $\mathcal{T}(P)$.

Let $(v_b, v_{b'})$ be the leftmost blocking diagonal of T. Consider a diagonal (v_1, v_j) of T. If $j \geq b$, then (v_1, v_j) is a generating diagonal of T. Therefore, according to Lemma 3.3.5, $T(v_1, v_j)$ is a child of T in $\mathcal{T}(P)$. Thus, for all diagonals (v_1, v_j) of T such that $j \geq b$, a new triangulation is generated by flipping (v_1, v_j).

If $j < b$, then (v_1, v_j) is a non-generating diagonal of T and according to Lemma 3.3.5, we can not flip (v_1, v_j) to generate a new triangulation from T.

Based on the case analysis above, we can generate all triangulations of a convex polygon P of n vertices. The algorithm is as follows.

Procedure find-all-child-triangulations(T)

begin

output T; {output the difference from the previous triangulation}

if T has no generating diagonals **then return** ;

Let $(v_b, v_{b'})$ be the leftmost blocking diagonal of T;

for all $j \geq b$

if (v_1, v_j) is a diagonal of T **then**

find-all-child-triangulations($T(v_1, v_j)$); {Case 2}

end;

Algorithm find-all-triangulations(n)

begin

output *root* T_r;

$T = T_r$;

for $j = n - 1$ **to** 3

find-all-child-triangulations($T(v_1, v_j)$); {Case 1}

end.

The following theorem describes the correctness and performance of the algorithm **find-all-triangulations**.

Theorem 3.3.7 *Given a convex polygon P of n vertices, the algorithm* ***find-all-triangulations***

generates all the triangulations of P in $O(1)$ time per triangulation on average, without duplications and omissions. The space complexity of the algorithm is $O(n)$.

Proof. Let T be a triangulation of P and $T(v_1, v_j)$ be the triangulation generated from T by flipping the diagonal (v_1, v_j) of T. The algorithm **find-all-triangulations** generates $T(v_1, v_j)$ from T if only if (v_1, v_j) is a generating diagonal of T. Therefore, according to Lemma 3.3.5, T is the parent of $T(v_1, v_j)$. That means, each triangulation T of P is generated from its parent only; therefore, duplication can not occur. To prove that no omission occurs, we use Lemma 3.3.2. Lemma 3.3.2 implies that for any triangulation T of P, there is a unique path from the root T_r to T in $\mathcal{T}(P)$. Thus, to show that the algorithm **find-all-triangulations** does not omit any triangulation, it is sufficient to prove that the algorithm **find-all-triangulations** generates all the children of a triangulation T. By Lemma 3.3.5, to generate the children of a triangulation T, only the generating diagonals of T need to be flipped. Since the algorithm **find-all-triangulations** flips all the generating diagonals of a triangulation T to generate new triangulations from T, all the children of T in $\mathcal{T}(P)$ are generated.

The complexity of the algorithm can be found as follows. We need to store the generating set C for the current triangulation T of P. Since the maximum cardinality of C is $n-3$, it take $O(n)$ space to store it. Along with C, we need to maintain for T, the set of opposite pairs O and update it while generating children. We also need to maintain another list L for listing the diagonals of T. To generate the triangulations of P, we start at the root of $\mathcal{T}(P)$. For the root of $\mathcal{T}(P)$, C is identical to L and C can be found in $O(n)$ time. When a generating diagonal of a triangulation T is flipped, that diagonal is replaced in the list L of T by its opposite pair in T to get the list L of the child. Since we use a recursive procedure to generate the triangulations without constructing the whole $\mathcal{T}(P)$, and the depth of the tree is $n-2$ (number of diagonals in the root plus one), the algorithm uses $O(n)$ space.

Now the question is how can we update C and O? By implementing these two sets

using linked lists and storing appropriate pointers at each node on the path from the root of $T(P)$ to the current triangulation T, we can do it in constant time. Let (v_1, v_j) be the diagonal of T to be flipped. The updated lists C and O correspond to the newly generated child of T.

Flipping the generating diagonal (v_1, v_j) of T can change the opposite pairs of maximum two other candidate diagonals of T in the representation of $T(v_1, v_j)$. In our algorithm, we only need to change the opposite pairs of candidate diagonals of $T(v_1, v_j)$. Let $(v_1, v_i), (v_1, v_j)$ and (v_1, v_k) be three candidate diagonals of T, $k < j < i$, such that $\langle v_1, v_k, v_j, v_i \rangle$ is a convex quadrilateral of T, as shown in Figure 3.12. We now flip (v_1, v_j) of T to generate the child $T(v_1, v_j)$ of T. Flipping the diagonal (v_1, v_j) of T changes the opposite pairs of the diagonals (v_1, v_i) and (v_1, v_k) of T in $T(v_1, v_j)$. The changes can be done as follows.

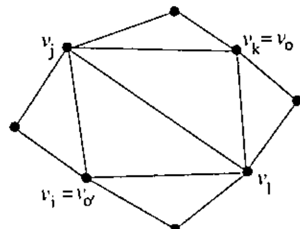


Figure 3.12: Flipping (v_1, v_j) can affect two candidate diagonals.

Let $(v_o, v_{o'})$ be the opposite pair of (v_1, v_j) in T. Here $o = k$ and $o' = i$, as shown in Figure 3.12. Let the opposite pair of (v_1, v_i) in T be $(v_l, v_{l'})$. Then $l = j$ and the opposite pair of (v_1, v_i) in $T(v_1, v_j)$ is $(v_o, v_{l'})$. Similarly, if the opposite pair of (v_1, v_k) is $(v_s, v_{s'})$ in T, then $s' = j$ and the opposite pair of (v_1, v_k) in $T(v_1, v_j)$ will be $(v_s, v_{o'})$. Figure 3.13 shows the update operations. Clearly, these updates can be done in $O(1)$ time.

Thus, if a triangulation T has k children, all of them can be generated in $O(k)$ time. Therefore each child of T is generated in $O(1)$ time on average. □

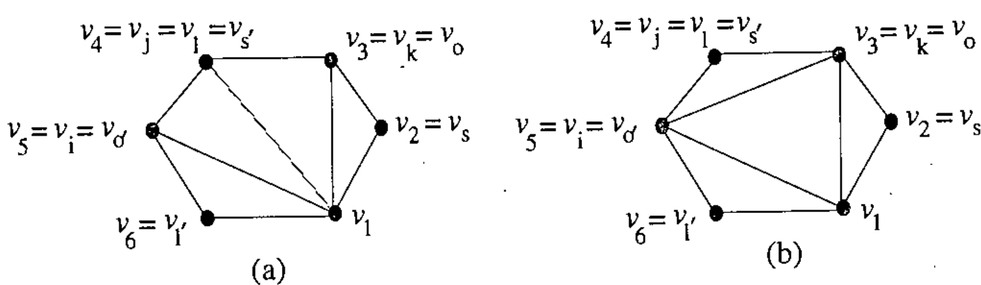


Figure 3.13: Illustration of update operations for opposite pairs of two affected edges; (a) parent and (b) child.

3.4 Unlabeled Triangulations of a Convex Polygon

In this section we modify our algorithm for generating all labeled triangulations of a convex polygon P of n vertices to generate unlabeled triangulations of P.

Generating unlabeled triangulations of a convex polygon P is more difficult than generating labeled triangulations; if vertices of P are not numbered then there arise "rotational" and "mirror repetitions" among the triangulations of P. Two unlabeled triangulations of a convex polygon are *rotationally equivalent* to each other, if one can be found by rotating the other one, when the labels are removed. Similarly, two unlabeled triangulations of a convex polygon are *mirror image* of each other, if one can be found by taking the mirror image of the other one. For example, the triangulations of Figure 3.14(a) and (b) are rotationally similar if we remove the labels. The two triangulations of Figure 3.15 are mirror images of the one another if no labels are used. In this section, we modify our algorithm for generating all triangulations of a convex polygon to avoid such repetitions. The main idea of the modified algorithm is to consider each triangulation of P as belonging to a particular class. Those triangulations of P which are rotationally equivalent or mirror images of one another, forms a class of triangulations. We choose one particular triangulation from each class as the representative of that class. The modified algorithm still uses the labels while generating the triangulations, but avoids any rotational or mirror repetitions by outputting a triangulation only if it is the representative of a particular

class. Thus, our modified algorithm constructs the tree of triangulations T_6 of a convex polygon of six vertices as shown in Figure 3.16. Note that, only 3 triangulations are there in Figure 3.16, while the tree of triangulations of Figure 3.2 contains 14 triangulations.

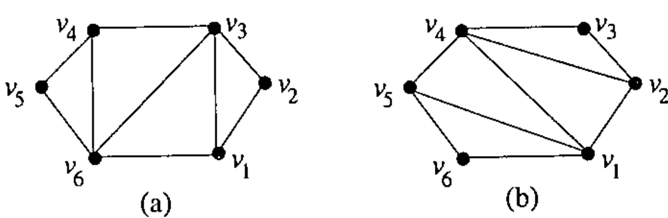


Figure 3.14: Triangulations of (a) and (b) are rotationally equivalent when the labels are removed.

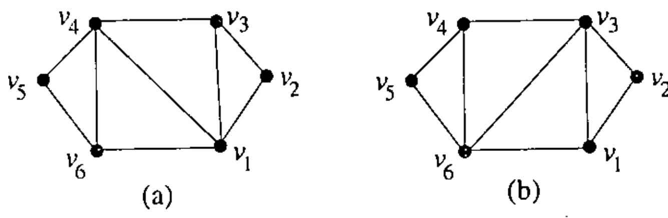


Figure 3.15: Triangulations of (a) and (b) are are mirror image of each other when the labels are removed.

We now give a new representation of each triangulation of a convex polygon that enables us to avoid any rotational or mirror repetitions easily. Let T be a triangulation of P where the vertices of P are labeled sequentially from v_1 to v_n. A *labeled degree sequence* $\langle d_1, d_2, \cdots, d_n \rangle$ of T is the sequence of degrees of the vertices, where d_i is the degree of v_i in the graph associated with T. A vertex with degree 2 is called an *ear* of T. We thus have the following lemma.

Lemma 3.4.1 *Let T be a labeled triangulation of a convex polygon P of n vertices. Then T can be represented uniquely by its labeled degree sequence.*

Proof. Let $\langle d_1, d_2, \cdots, d_n \rangle$ be the labeled degree sequence of T. We note that T has at least two ears. Let v_i is the clockwise first ear. Remove it and decrease the degrees of its

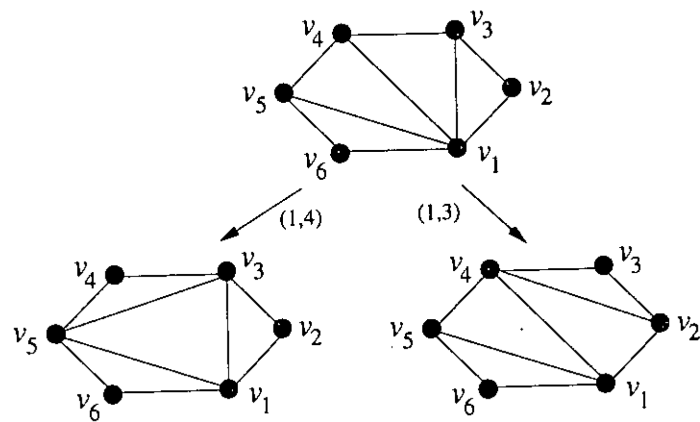


Figure 3.16: Illustration of genealogical tree of six vertices when rotational and mirror repetitions are not allowed.

two neighboring vertices by one. Apply the procedure recursively until the vertices v_1 ans v_2 are left. Thus we get a sequence of vertices $v_{i_1}, v_{i_2}, \cdots v_{i_{n-2}}$. Now adding the vertices in reverse order we can generate T. Thus there is a bijection between the triangulations of a convex polygon and the labeled degree sequences of the triangulations. □

3.4.1 Removing Rotational Repetitions

First we describe the algorithm for avoiding rotational repetitions. The following fact is very important for that purpose.

We have the following idea to avoid rotational repetitions. One can observe the following fact.

Fact 3.4.2 *Let T and T' be two triangulations of a convex polygon P of n vertices, which are rotationally equivalent to each other. Then, by rotating the labeled degree sequence of T, we get the labeled degree sequence of T'.*

As an illustration of the Fact 3.4.2, the triangulations of Figure 3.14(a) and (b) have the labeled degree sequences $\langle 3, 2, 4, 3, 2, 4 \rangle$ and $\langle 4, 3, 2, 4, 3, 2 \rangle$ respectively. By right rotating

the labeled degree sequence of the triangulation of Figure 3.14(a) 4 times, we get the labeled degree sequence of the triangulation of Figure 3.14(b).

Let T and T' be two triangulations of a convex polygon P of n vertices, which are rotationally equivalent to each other. Let $\langle d_1, d_2, \cdots, d_n \rangle$ and $\langle d'_1, d'_2, \cdots, d'_n \rangle$ be the labeled degree sequences of T and T' respectively. Let $d_1 = d'_1, d_2 = d'_2, \cdots, d_{k-1} = d'_{k-1}$ and $d_k > d'_k$ for some k, $1 \leq k \leq n$. We say that the sequence $\langle d_1, d_2, \cdots, d_n \rangle$ is *greater* than the sequence $\langle d'_1, d'_2, \cdots, d'_n \rangle$ and T has a *greater* sequence than T'. For example, the triangulations of Figure 3.17(a) and (b) have the degree sequences $\langle 5, 2, 5, 2, 3, 4, 3, 2 \rangle$ and $\langle 4, 2, 3, 4, 3, 3, 2, 5 \rangle$ respectively and the first sequence is greater than the second one. Thus the triangulation of Figure3.17(a) is greater than the triangulation of Figure 3.17(b).

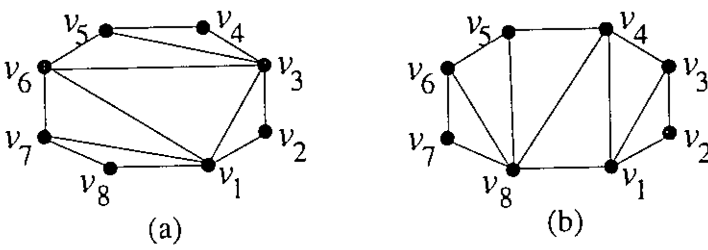


Figure 3.17: Illustration of two triangulations where one has greater degree sequence.

Let S be the set of those triangulations of a convex polygon of n vertices, where the triangulations are rotationally equivalent to each other. Let T be the triangulation in S whose degree sequence is greater than all other triangulations in S. Then, the labeled degree sequence of T is the *canonical representation* of S. We say that T has the *greatest* labeled degree sequence and T is the *representative* of S. We modify our algorithm to avoid any rotational repetitions as follows. We output each triangulation T only if it has the greatest labeled degree sequence. Let $\langle d_1, d_2 \cdots d_n \rangle$ be the degree sequence of T. If $d_1 > d_i$ for $2 \leq i \leq n$, then T has the greatest labeled degree sequence. This can be found in $O(1)$ time as explained later. Otherwise, we generate $n-1$ other degree sequences by right rotating T's degree sequence and check whether T's sequence is greater. In this case, it takes $O(n^2)$ time to find whether T has the greatest labeled degree sequence.

For the triangulation T, we need to maintain an array D to store the degree sequence. It takes $O(n)$ space. Let (v_1, v_j) is a generating diagonal in T with opposite pair $(v_o, v_{o'})$. Flipping (v_1, v_j) changes the degrees of four vertices. The degrees of v_1 and v_j are reduced by one and the degrees of v_o and $v_{o'}$ are increased by one. All these updates can be done in $O(1)$ time. Let $\langle d'_1, d'_2, \cdots, d'_n \rangle$ be the resultant degree sequence and T' is the new triangulation. We can easily check whether $d'_1 > d'_i$ for $2 \leq i \leq n$ by storing the highest degree d_{max} among nodes other than v_1 and updating it while generating a new triangulation. Now there are three cases.

Case 1: If $d'_1 > d_{max}$, then output T'.

Case 2: If $d'_1 = d_{max}$, then check whether T' has the greatest labeled degree sequence. If YES, then output T'.

Case 3: If $d'_1 < d_{max}$, then ignore T' and prune the subtree of triangulations rooted at T'.

Let S be the set of those triangulations of a convex polygon P of n vertices, where the triangulations are rotationally equivalent to each other. Let T be the triangulation in S whose degree sequence is greater than all other triangulations in S. Then, the labeled degree sequence of T is the *canonical representation* of S. We say that T has the *greatest* labeled degree sequence and T is the *representative* of S. To avoid any rotational repetitions among the triangulations of P, we output each triangulation T of P only if it has the greatest labeled degree sequence. Let $\langle d_1, d_2 \cdots d_n \rangle$ be the degree sequence of T. If $d_1 > d_i$ for $2 \leq i \leq n$, then T has the greatest labeled degree sequence. This can be found in $O(1)$ time as explained later. Otherwise, we generate $n - 1$ other degree sequences by right rotating T's degree sequence and check whether T's sequence is greater. In this case, it takes $O(n^2)$ time to find whether T has the greatest labeled degree sequence. We also need to maintain for T an array to store the degree sequence. It takes $O(n)$ space.

3.4.2 Avoiding Mirror Repetitions

We now further modify our algorithm of generating all triangulations of a convex polygon to avoid any mirror image repetitions.

Let T be a triangulation of an n vertex convex polygon having labeled degree sequence $\langle d_1, d_2, \cdots, d_n \rangle$. Assume that T has the greatest labeled degree sequence than all other triangulations which are rotationally similar to T. Let T' be the triangulation which is the mirror image of T. Using the following fact we can find the labeled degree sequence of T'.

Fact 3.4.3 *Let T and T' be two triangulations of a convex polygon of n vertices, which are mirror images of each other. Let T has the labeled degree sequence $\langle d_1, d_2, \cdots, d_n \rangle$. Then the labeled degree sequence of T' is $\langle d_n, d_{n-1}, \cdots, d_2, d_1 \rangle$.*

For example, the triangulation of Figure 3.15(a) has the degree sequence $\langle 4, 2, 3, 4, 2, 3 \rangle$. The triangulation of Figure reffig:mirrorimage(b), which is the mirror image of the triangulation of Figure 3.15(a), has the *reverse* degree sequence $\langle 3, 2, 4, 3, 2, 4 \rangle$.

Now, using the labeled degree sequence of T', we can avoid mirror image repetitions as follows. We start with the sequence $\langle d_n, d_{n-1}, \cdots, d_2, d_1 \rangle$, and from it we generate $n-1$ other sequences by right rotation. These $n-1$ sequences corresponds to all the triangulations which are rotationally similar to T'. We compare the degree sequence of T with all these sequences to determine whether T's sequence is the greatest. Thus, we have to compare T's sequence with a total of n sequences. This takes $O(n^2)$ time. If T's sequence is found greater than all these sequences, then we output T. Otherwise we discard T and prune the subtree rooted at T. Since all we need is to store the sequence of T, the space complexity is $O(n)$.

Thus we have the following theorem.

Theorem 3.4.4 *For a convex polygon P of n vertices, all the triangulations of P can be*

found in time $O(n^2)$ per triangulation, where the vertices of P are not numbered, avoiding the rotational and mirror image repetitions. The space complexity is $O(n)$.

3.5 Conclusion

In this chapter we gave two algorithms. The first one generates all labeled triangulations of a convex polygon P of n vertices. In this case all the vertices of P are numbered sequentially. The second algorithm generates unlabeled triangulations of a convex polygon P of n vertices, where the vertices of P are not numbered and avoids rotational and mirror image repetitions among the triangulations of P.

The main idea behind the algorithms was to generate each triangulation from previous one by making a constant amount of local changes. For that purpose we defined a tree structure among the triangulations of P. The idea can be traced back to the well known technique called *Combinatorial Gray Code Approach*, although the main feature of our algorithm is the data structure we used to represent each triangulation. That data structure is crucial in developing the algorithm for generating all triangulations of a given plane graph, as described in the next chapter. Our algorithm for generating all labeled triangulations of a convex polygon P of n vertices generates each new triangulation in $O(1)$ time and uses $O(n)$ space. The algorithm for generating unlabeled triangulations is based on the algorithm for labeled triangulations and generates each triangulation in $O(n^2)$ time with linear space complexity.

Chapter 4

Triangulations of Plane Graphs

4.1 Introduction

In this chapter we present the main result of this thesis, an algorithm for generating all triangulations of a biconnected plane graph G. Our algorithm for generating all triangulations of a biconnected plane graph G is based on the algorithm for generating all triangulations of a convex polygon P of n vertices as described in Chapter 3. Here also we define a parent-child relationship among the triangulations of G and denote the corresponding genealogical tree of G by $\mathcal{T}(G)$. To make the material accessible, we describe the algorithm in two parts. We first handle the case where G is a biconnected outerplanar graph and describe in detail the algorithm for generating all triangulations of G. Later we generalize the algorithm for generating all triangulations of a biconnected outerplanar graph to generate all triangulations of a biconnected plane graph.

The rest of the chapter is organized as follows. In Section 4.2 we give an algorithm that generates all triangulations of a biconnected outerplanar graph G of n vertices. Section 4.3 describes the algorithm for generating all triangulations of a biconnected plane graph. Finally, Section 4.4 is the conclusion.

4.2 Algorithm for Biconnected Outerplanar Graphs

Let G be a biconnected outerplanar graph of n vertices. The genealogical tree of the biconnected outerplanar graph G of Figure 4.1(a) is shown in Figure 4.1(b).

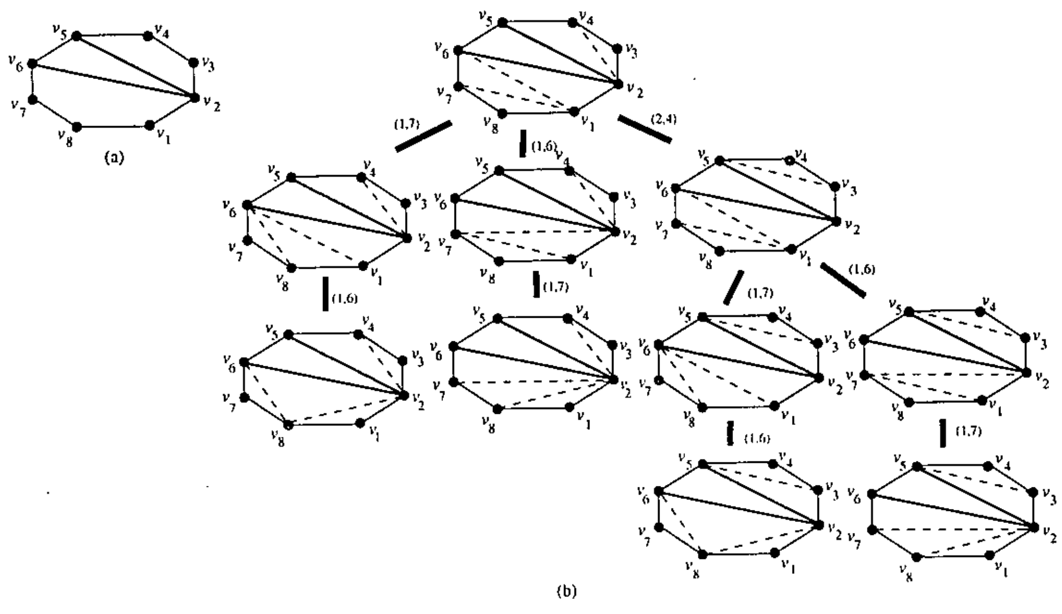


Figure 4.1: Illustration of (a) an outerplanar graph G of eight vertices and (b) genealogical tree $\mathcal{T}(G)$ of G.

Let F be an inner face of a biconnected outerplanar graph G, where the boundary of F contains s vertices. Then the boundary of the face F can be drawn as the boundary of a convex polygon P of s vertices and a triangulation of P corresponds to a triangulation of the face F of G. For example, the face $\langle v_1, v_2, v_5, v_6, v_7 \rangle$ of the graph in Figure 4.2(a) can be drawn as the convex polygon P in Figure 4.2(b). Then the triangulation of P in Figure 4.2(c) corresponds to the triangulation of the face $\langle v_1, v_2, v_5, v_6, v_7 \rangle$ of the graph of Figure 4.2(a), as shown in Figure 4.2(d). Our idea is to apply the algorithm for generating all triangulations of a convex polygon to each of the inner faces of G. By finding the triangulations of the inner faces of G and then combining those triangulations we generate the triangulations of G itself.

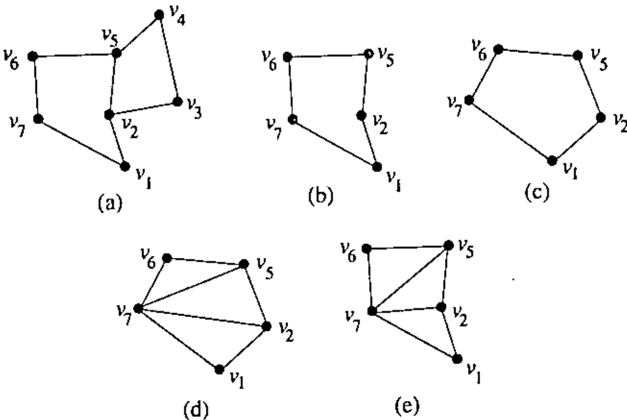


Figure 4.2: An inner face of a biconnected outerplanar graph can be viewed as a convex polygon.

4.2.1 Finding the Root

In this section we give an algorithm to find the root triangulation of the genealogical tree $\mathcal{T}(G)$ of a biconnected outerplanar graph G of n vertices.

Finding a root for $\mathcal{T}(G)$ is more difficult than finding the root for $\mathcal{T}(P)$. The idea is to treat each inner face F_i of G as a convex polygon P_i and find the root triangulation of the genealogical tree of P_i using the definition of root triangulation of Section 3.3.2. From the root triangulation of the genealogical tree of P_i we get a triangulation of F_i. By combining the triangulations found for each face F_i of G this way, we get a triangulation of the graph G. We take this specific triangulation as the root triangulation T_r of the genealogical tree $\mathcal{T}(G)$ of G. Figure 4.3(a) shows a biconnected outerplanar graph G of 10 vertices. Corresponding root in $\mathcal{T}(G)$ is shown in Figure 4.3(b).

Assume that the biconnected outerplanar graph G has k faces, labeled $F_1, F_2, \cdots, F_k$. We say that the face F_i *precedes* the face F_j whenver $i < j$. For each face F_i of G, there is a convex polygon P_i *associated* with F_i, where the number of vertices of P_i and F_i are same and the vertices of P_i are labeled similar to the vertices of F_i. The face F_i has the candidate set C_i, where C_i is the generating set of the root triangulation of the genealogical tree of P_i. The generating set C^o of G is the *ordered union* of $C_1, C_2, \cdots, C_k$. That is

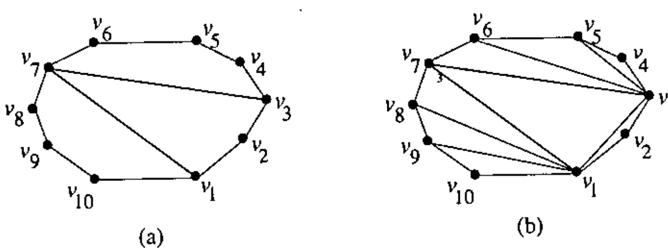


Figure 4.3: Illustration of (a) a biconnected outerplanar graph of 10 vertices and (b) corresponding root in $\mathcal{T}(G)$

$C^o = C_1 \cup C_2 \cup \cdots \cup C_k$ and the edges of C_i are listed in C^o before the edges of C_j, whenever $i < j$. For example, in Figure 4.3(b), $C_1 = \{(v_1, v_9), (v_1, v_8)\}$, $C_2 = \{(v_1, v_3)\}$, and $C_3 = \{(v_1, v_6), (v_1, v_5)\}$. Thus $C^o = C_1 \cup C_2 \cup C_3 = \{(v_1, v_9), (v_1, v_8), (v_1, v_3), (v_1, v_6), (v_1, v_5)\}$.

We traverse the face F_i of G to find the generating set C_i of F_i. We traverse the face F_i of G using the doubly connected adjacency list representation of G [NR04]. Face F_i can be traversed in time proportional to the number of vertices on the boundary of it. Assume that we traverse the face F_i clockwise starting at vertex v_j. We find the candidates edges of F_i and corresponding opposite pairs as follows. Let v_i be a vertex on the boundary of the face F_i. The edge (v_j, v_i) is added to the generating set C_i of F_i if and only if (v_j, v_i) is not an edge of G. Thus we have the following lemma.

Lemma 4.2.1 *Let G be a biconnected outerplanar graph of n vertices. Then the root triangulation of the genealogical tree $\mathcal{T}(G)$ of G can be found in $O(n)$ time.*

Proof. In a biconnected outerplanar graph G of n vertices, the maximum number of edges is $2n - 3$. The number of inner faces of G is at most $n - 2$. Since each face F_i of G can be traversed in time proportional to the number of edges on the boundary of F_i and each edge can be shared by at most two faces of G, traversing all the faces of G requires time proportional to the total number of edges of G. Thus the root of $\mathcal{T}(G)$ can be found in $O(n)$ time. □

4.2.2 The Child Generation Rule

In this section we give the rules for generating child triangulations of a triangulation T of a biconnected outerplanr graph G.

Let F_i be a face and T be a triangulation of G. The current triangulation of the face F_i in T corresponds to a triangulation of the convex polygon P_i associated with F_i. The generating diagonals of the current triangulation of P_i are called the *candidate edges* of F_i in T and the leftmost blocking diagonal of the current triangulation of P_i is called the *blocking edge* of F_i in T. If there is no generating diagonals in the current triangulation of P_i, then F_i has no candidate edges in T. The candidate edges of F_i in T are called *generating edges* of T if there is no blocking edge in any of the faces F_j in T, $j < i$. We call the face F_i a *generating face* of T if F_i contains generating edges. To generate a child triangulation of a triangulation T of G, we find a generating face F of T and then flip a candidate edge of F.

A generating face F in a triangulation T of G can be found as follows. Let T be the root triangulation of $\mathcal{T}(G)$. Then all the faces of G are generating faces of T. Otherwise, assume that T is generated from its parent by flipping a candidate edge of the face F_i of G. Then all the faces F_j, $j \leq i$ are the generating faces of T.

The above child generation rule ensures that each child of a triangulation T of G can be generated in $O(1)$ time. For the triangulation T, we maintain three lists; the list of candidate edges C^o, the list of opposite pairs O^o, and the list of edges L^o representing T. It requires $O(n)$ space to store the lists. Updating the lists are similar to the procedures explained in Section 3.3. Note that, flipping an edge of face F_i affects the candidate edges of the face F_i only.

Thus we have the following theorem.

Theorem 4.2.2 *Let G be a biconnected outerplanar graph of n vertices. Then the children of a triangulation T in $\mathcal{T}(G)$ can be generated in time $O(1)$ per triangulation, with*

$O(n)$ space complexity.

4.3 Algorithm for Biconnected Plane Graphs

The algorithm for generating all triangulations of a biconnected outerplanar graph can be readily extended to a biconnected plane graph. Let G be a biconnected plane graph of n vertices. Given the doubly connected adjacency list representation of G, we can find the root triangulation of the genealogical tree of G in $O(n)$ time. We maintain the generating set similarly as in biconnected outerplanar graph. Generating the new triangulations from a triangulation of the biconnected plane graph is similar to the biconnected outerplanar graph and cases are also same. Since the number of edges in any plane graph is bounded by a linear function of n, the space complexity is also linear.

While generating triangulations of plane graphs, we have to triangulate the outer face also. This can be done easily, since triangulating the outer face is similar to triangulating an inner face. For example, boundary of the outer face of the graph of Figure 4.4(a) is shown in Figure 4.4(b) and Figure 4.4(c) is one possible triangulation of that outer face. We can consider the outer face as an inner face and the triangulation of Figure 4.4(d) corresponds to the triangulation of Figure 4.4(c).

Thus we have the following theorem.

Theorem 4.3.1 *Given a biconnected plane graph G of n vertices, the algorithm for generating all triangulations of G generates each triangulation in $O(1)$ time, with $O(n)$ space complexity.*

4.4 Conclusion

In this chapter we gave algorithms that generate all triangulations of a biconnected outerplanar graph and a biconnected plane graph. Both of these algorithms are based on

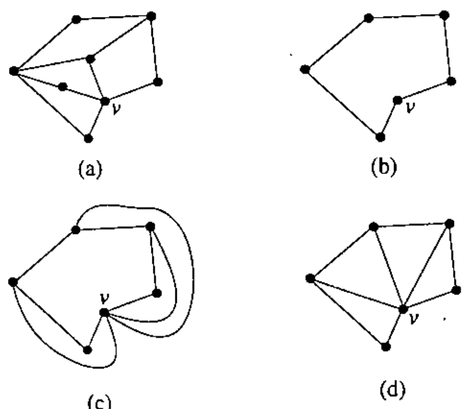


Figure 4.4: Triangulating outer face of a plane graph; (a) the graph, (b) boundary of the outer face, (c) one triangulation of the outer face and (d) equivalent triangulation of the inner face.

the algorithm we described in Chapter 3 that generates all labeled triangulations of a convex polygon. The algorithms described in this chapter requires some preprocessing of the given graph G to find the root of the tree of triangulations or the genealogical tree of G. Starting at the root triangulations, all other triangulations of G are generated where each triangulation is generated from previous one in $O(1)$ time using only $O(n)$ space, where n is the number of vertices of G.

Chapter 5

Conclusion

This thesis deals with algorithms for generating all triangulations of a plane graph. We have given efficient algorithms to generate all triangulations of a convex polygon P of n vertices and based on that algorithm developed algorithms for generating all triangulations of a biconnected outerplanar graph and biconnected plane graph.

We first summarize each chapter and its contributions. In Chapter 1 we have given a brief description of the problem we have addressed in this thesis and discussed our motivations behind solving the problem. We also have described the main algorithmic challenges that any enumerations algorithm has to face and reviewed some of the existing literature.

In Chapter 2 we have introduced graph theoretical terminologies that have been used throughout this thesis.

In Chapter 3 we have given two algorithms. The first algorithm generates all triangulations of a convex polygon P of n vertices where the vertices of P are labeled sequentially. The algorithm generates each triangulation of P in $O(1)$ time per triangulation and uses $O(n)$ space. The other algorithm of this chapter generates all unlabeled triangulations of P in worst case time $O(n^2)$ per triangulation using only linear space.

In Chapter 4 we have given algorithms that generate all triangulations of a biconnected

outerplanar graph and a biconnected plane graph. The algorithms are based on the algorithm for generating all triangulations of a convex polygon P of n vertices given in Chapter 3. The algorithms in this Chapter generate each triangulation of a biconnected outerplanar graph or a plane graph G of n vertices in time $O(1)$ per triangulation with liner space complexity.

The following problems related to the generation of triangulations of graphs and polygons are still open.

1. Develop an algorithm that generates all triangulations of a connected plane graph.
2. Is there any constant time algorithm that generates unlabeled triangulations of a convex polygon?
3. Develop an algorithm that generates all triangulations of a simple polygon.

Bibliography

[Avi96] D. Avis, *Generating rooted triangulations without repetitions*, Algorithmica, 16, pp. 618-632, 1996.

[AF96] D. Avis and K. Fukuda, *Reverse search for enumeration*, Discrete Applied Mathematics, 6, pp. 82-90, 1996.

[Aic99] O. Aichholzer, *The path of a triangulation*, Proc. of 15th Annual Symposium on Computational Geometry, pp. 14-23, 1999.

[Bes02] S. Bespamyatnikh, *An efficient algorithm for enumeration of triangulations*, Computational Geometry: Theory and Applications, 23, pp. 271-279, 2002.

[BV04] J. Baril, V. Vajnovszki, *Gray code for derangements*, Discrete Applied Mathematics, 140, pp. 207-221, 2004.

[Cha91] B. Chazelle, *Triangulating a polygon in linear time*, Discrete Computational Geometry, 6, pp. 485-524, 1991.

[CLR90] T. M. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[DVOS00] M. de Berg, M. van Krevald, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2000.

[ES94] P. Epstein, J.-R Sack, *Generating triangulations at random*, ACM Transaction on Modeling and Computer Simulation, 4, pp. 267-278, 1994.

[For87] S. J. Fortune, *A note on Delaunay diagonal flip*, Manuscript, AT&T Bell Lab, Murray Hill, NJ, 1987.

[HN99] F. Hurtadò and M. Noy, *Graph of triangulations of a convex polygon and tree of triangulations*, Computational Geometry 13, pp. 179-188, 1999.

[HNU99] F. Hurtado, M. Noy and J. Urrutia, *Flipping diagonals in triangulations*, Discrete Computational Geometry 22, pp. 333-346, 1999.

[HOS96] S Hanke, T. Ottmann and S. Schuierer, *The edge-flipping distance of triangulations*, Journal of Universal Computer Science, 2, pp. 570-579, 1996.

[Joh63] S. M. Johson, *Generation of permuatations by adjacent transpositions*, Mathematics of Computation, 17, pp. 282-285, 1963.

[JWW80] J. T. Joichi, D. E. White, and S. G. Williamson, *Combinatorial gray codes*, SIAM Journal on Computing, 9(1), pp. 130-141, 1980.

[KNN99] H. Komuro, A. Nakamoto and S. Negami, *Diagonal flips in triangulations on closed surfaces with minimum degree at least 4*, Journal of Combinatorial Theory, Series B, 76, pp. 68-92, 1999.

[Lee89] C. W. Lee, *The associahedron and triangulations of the n-gon*, European Journal of Combinatorics, 10, pp. 551-560, 1989.

[LN01] Z. Li and S. Nakano, *Efficient generation of plane triangulations without repetitions*, Proc. of ICALP 2001, LNCS 2076, pp. 433-443, 2001.

[Mck98] B. D. Mckay, *Isomorph-free exhaustive generation*, Journal of Algorithms, 26, pp. 306-324, 1998.

[Nak02] S Nakano, *Efficient generation of plane trees*, Information Processing Letters, 84, pp. 167-172, 2002.

[NR04] T. Nishizeki and M. S. Rahman, *Planar Graph Drawing*, World Scientific, 2004.

[NU04a] S. Nakano and T. Uno, *More efficient generation of plane triangulations*, Proc. of GD 2003, LNCS 2912, pp. 273-292, 2004.

[NU04b] S. Nakano and T Uno, *Constant time generation of trees with specified diameter*, Proc. of WG 2004, pp. 33-45, 2004.

[Rou98] J O'Rourke, *Computational Geometry in C*, second edition, Cambridge University Press, 1998.

[SY99] S. M. Sait and H. Youssef, *VLSI Physical Design Automation: Theory and Practice*, World Scientific, 1999.

[Sav97] C. Savage, *A survey of combinatorial gray codes*, SIAM Review, 39, pp. 605-629, 1997.

[STT88] D. Sleator, R. Tarjan and W. Thurston, *Rotation distance, triangulations, and hyperbolic geometry*, Journal of American Mathematical Society, 1, pp. 647-681, 1988.

[Tro62] H. F. Trotter, *PERM (Algorithm 115)*, Communications of the ACM, 5, pp. 434-435, 1962.

[Wes01] D. B. West, *Introduction to Graph Theory, 2nd Ed*, Prentice Hall, 2001.

Index

$E(G)$, 19

$O(n)$, 27

$V(G)$, 19

grid drawing, 4

algorithm, 26
- complexity, 26
- exponential, 27
- linear time, 27
- polynomial, 27
- running time, 27

Art Gallery Problem, 8

blocking edge, 64

candidate edges, 64

canonical ordering, 4

canonical representation, 56

Catalan Family, 34

combinatorial gray code, 32
- Johnson-Trotter algorithm, 33

complexity
- time, 10

cycle, 24

degree sequence, 26
- greatest, 56
- labeled, 54

depth first search, 28

DFS , *see* depth first search

diagonal
- blocking, 39
  - leftmost, 42
- candidate, 43
- generating, 44
- non-generating, 44

duplication, 10

ear, 54

edge, 19

face traversal, 29
- algorithm, 30

family tree, 34

flipping, 38

genealogical tree, 38
- root, 39

generating face, 64

- generating set, 44
- graph, 19
  - connectivity, 20
  - edges, 19
  - face, 20
    - inner face, 20
    - outer face, 20
    - precede, 62
  - outerplanar, 24
  - planar, 20
  - plane, 20
    - data structure, 29
  - simple, 19
  - triangulation, 23
  - vertex, 19
    - degree, 20
- graph drawing, 4
- graph traversal, 28

- I/O operation, 10

- mirror image, 53

- NP hard, 9

- opposite pair, 48

- path, 24
- polygon, 18
  - convex, 19
  - diagonal, 19
  - edge, 18
  - exterior, 19
  - interior, 19
  - representation, 19
  - side, 18
  - simple, 19
  - triangulation, 21
  - vertex, 18
    - visible, 19

- representation of triangulations, 48
- rotationally equivalent, 53

- table
  - comparison, 17
- tree, 24
  - ancestor, 25
  - binary, 26
  - child, 25
  - depth, 26
  - descendant, 25
  - height, 26
  - leaf, 25
  - level, 26
  - node, 24
    - internal, 25
  - parent, 25

প্রকৌশল বিশ্ববিদ্যালয়
নং...............
তাং...............
লাইব্রেরী, ঢাকা

root, 25

triangulations

based plane, 14

convex polygon, 13

graph of, 12

labeled, 16, 22

tree of, 13

unlabeled, 16, 22

vision

blocked, 40

clearer, 41

full, 39

VLSI floorplan, 7

walk, 24